Grammar Repair with Examples and Tree Automata

YUNJEONG LEE, National University of Singapore, Singapore
GOKUL RAJIV, National University of Singapore, Singapore
ILYA SERGEY, National University of Singapore, Singapore

Context-free grammars (CFGs) are the de-facto formalism for declaratively describing concrete syntax for programming languages and generating parsers. One of the major challenges in defining a desired syntax is ruling out all possible ambiguities in the CFG productions that determine scoping rules as well as operator precedence and associativity. Practical tools for parser generation typically apply ad-hoc approaches for resolving such ambiguities, which might result in a parser's behaviour that contradicts the intents of the language designer. In this work, we present a user-friendly approach to soundly *repair* grammars with ambiguities, which is inspired by the *programming by example* line of research in automated program synthesis. At the heart of our approach is the interpretation of both the initial CFG and additional examples that define the desired restrictions in precedence and associativity, as *tree automata* (TAs). The technical novelties of our approach are (1) a new TA learning algorithm that constructs an automaton based on the original grammar and examples that encode the user's preferred ways of resolving ambiguities all in a single TA, and (2) an efficient algorithm for TA intersection that utilises reachability analysis and optimizations that significantly reduce the size of the resulting automaton, which results in idiomatic CFGs amenable to parser generators. We have proven the soundness of the algorithms, and implemented our approach in a tool called Greta, demonstrating its effectiveness on a series of case studies.

CCS Concepts: • **Theory of computation** → **Grammars and context-free languages**; *Parsing*; Tree languages; • **Software and its engineering** → Programming by example.

Additional Key Words and Phrases: context-free grammars, parsing, tree automata, programming by example

ACM Reference Format:
Yunjeong Lee, Gokul Rajiv, and Ilya Sergey. 2026. Grammar Repair with Examples and Tree Automata. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 134 (April 2026), 27 pages. https://doi.org/10.1145/3798242

1 Introduction

In recent years, substantial progress has been made on automatically synthesising grammars and parsers for context-free languages [6, 8, 14, 22]. Nevertheless, writing a precise description of a programming language's concrete syntax still remains a challenging and difficult-to-automate task. The main challenge stems from the multitude of possibilities to introduce ambiguities in the interpretation of language strings as syntax trees when defining the language's context-free grammar (CFG). For instance, according to the following CFG of a language of arithmetic expressions

$$\mathsf{S} \longrightarrow \mathsf{S} + \mathsf{S} \mid \mathsf{S} * \mathsf{S} \mid (\mathsf{S}) \mid x \mid y \mid z$$

the string $x + y * z$ can be parsed both as $(x + y) * z$ and $x + (y * z)$, even though only one of those interpretations is typically desired by the language designers. Modern frameworks for LR(k) parser generation, such as yacc [16], Beaver [12], ScalaBison [9], and Menhir [30] can detect an overapproximation of such ambiguities in operator precedence, associativity, and nesting, in the

Authors' Contact Information: Yunjeong Lee, National University of Singapore, Singapore, yunjeong.lee@u.nus.edu; Gokul Rajiv, National University of Singapore, Singapore, grajiv@u.nus.edu; Ilya Sergey, National University of Singapore, Singapore, ilya@nus.edu.sg.

ACM 2475-1421/2026/4-ART134
https://doi.org/10.1145/3798242

form of so-called shift/reduce and reduce/reduce conflicts, reporting them to the users and even resolving them automatically. Even though this simplifies the development of a language's syntax, automated ambiguity resolution might result in an interpretation of the syntax that is different from what is implicitly envisioned by its designer.

To provide more control over operator precedence, associativity, and nesting, existing tools for parser generation offer mechanisms to specify these properties explicitly [24, 32], while standard compiler textbooks provide general strategies to describe a CFG to avoid ambiguities in the first place [17, 36]. That said, adopting tool-specific conventions and following "good practices" for structuring CFGs often leads to grammars that are difficult to understand and maintain. Even worse, should an ambiguity be introduced in a CFG, expert knowledge, both in the structure of the object language and in the workings of the parser generator tool is required to correctly resolve it.

A more *declarative* approach to resolve ambiguities in a context-free grammar has been proposed by Adams and Might [2] who suggested capturing the restrictions imposed on top of an ambiguous grammar in the form of *tree regular expressions* (TREs) that explicitly forbid undesired classes of parse trees. The approach of Adams and Might exploits the fundamental connection between CFGs, tree-regular expressions, and tree automata (TAs), using the latter language representation, which is closed under intersection, as a way to produce the *repaired* (*i.e.* ambiguity-free) version of the grammar. Unfortunately, writing a correct tree-regular expression that resolves an ambiguity is not an easier task for a non-expert than fixing the ambiguity directly in the grammar, as it requires one to have a good grasp of TRE semantics to design suitable restrictions. The goal of this work is to enhance the tree automata-based approach to grammar repair and provide a user-friendly and sound way to repair grammars from ambiguities by adopting a popular *programming by example* paradigm from the works on automated program synthesis [15, 18, 19, 27, 34, 37].

Key idea. Our novel approach, dubbed *grammar repair by example*, resolves ambiguities in a grammar by suggesting to the user pairs of examples that demonstrate mutually-exclusive ways to resolve parsing conflicts, asking the user to choose one of them, and using the chosen examples to generate a "fixed" version of the grammar. More specifically, our approach (a) converts an ambiguity detected as an LR(1) parser generation conflict, into a small set of concrete examples that are shown to the user in the form of parse tree alternatives. Out of those examples, (b) the user chooses the examples corresponding to the desired syntax of the language in question. The chosen examples are then (c) automatically converted into a grammar restriction in the form of a tree automaton, which is next (d) intersected with the automaton corresponding to the original grammar, thus eliminating the ambiguity. Finally, (e) the result of the intersection is converted back to the CFG form, which is then analysed for ambiguities again, and, in case any are detected, the steps (a)-(e) repeat.

Challenges. The technical problems that needed to be solved in order to implement this approach in practice had to do with designing algorithms for the steps (c) and (d). In particular, while step (c) appears to be a textbook automata learning task, standard Angluin-style algorithms [4, 5, 7] are unsuitable for our scenario, as they require construction of tree examples for *all* the alphabet symbols. In other words, such examples would have to collectively represent the *entire grammar* rather than its *subsets* that are directly involved in ambiguities—which is what we aim to provide to the user of our approach. Furthermore, using the standard definition of tree automata intersection from the existing approaches as a way to update the grammar with the learned "fixes" [2, 10] *as is* produces non-idiomatic grammars that are difficult to comprehend.

We addressed these challenges by implementing two novel algorithms: for tree automata learning from example sub-trees of an input "base" grammar (for the step (c)) and for TA intersection (for (d)). Finally, we implemented the described end-to-end grammar repair pipeline in a tool.

Contributions. In summary, we make the following contributions:

```
V = { stmt, decl, expr, ident }
Σ = { SEMI, IF, THEN, ELSE, PLUS, STAR, INT, LPAREN, RPAREN, TINT, EQ }
S = stmt
P =
  { stmt → decl SEMI                      expr → expr PLUS expr
    stmt → IF expr THEN stmt              expr → expr STAR expr
    stmt → IF expr THEN stmt ELSE stmt    expr → INT
    decl → TINT ident EQ expr             expr → LPAREN expr RPAREN
    ident → IDENT                         expr → ident              }
```

Fig. 1. An example CFG $\mathcal{G}$ with ambiguities.

- Our main conceptual contribution is *grammar repair by example*—a novel approach to automatically resolve ambiguities in context-free grammars following a simple input from the user regarding what parse (sub)trees are acceptable. The workings of our approach rely on a fundamental relation between context-free grammars and tree automata (Sec. 2).
- Our first technical contribution is a novel algorithm for passive learning of tree automata from positive/negative parse subtree examples and its soundness proof stating that the synthesised automaton indeed correctly accepts all positive and rejects all negative examples (Sec. 3.1).
- Our second technical contribution is a new algorithm for computing an intersection of tree automata tailored to regular tree grammars producing a result that can be rendered as an idiomatic CFG, along with the proof of its correctness (Sec. 3.2).
- We implemented our approach to grammar repair by example in a tool called Greta on top of Menhir—a framework for parser generators in OCaml [30]. Our evaluation on examples from undergraduate compiler classes, questions posed on StackOverflow, and grammars of real-world languages demonstrates utility and efficiency of our approach: it does indeed fix ambiguities, with minimal help from the user, in most of the case studies and does so quite fast (Sec. 4).

2 Overview

This section illustrates an example run of Greta, demonstrating how ambiguities in a CFG can be resolved through a lightweight interaction with the user, by learning the "fixes" in the form of tree automata (TA) from user-provided examples and subsequently updating the grammar by means of TA intersection. We start from a characteristic example of an initial input CFG with ambiguities (Sec. 2.1), explain how it is translated into a TA (Sec. 2.2), describe generation of a TA based on the user-specified tree examples and the input CFG (Sec. 2.3), and show how the ambiguities are resolved by intersecting the two TAs and translating the result back to a CFG (Sec. 2.4).

2.1 Context-Free Grammars and Ambiguities

A context-free grammar (CFG) is formally defined as a tuple (V, Σ, S, P), where V is a set of nonterminal symbols, Σ a set of terminal symbols, $S \in V$ a start nonterminal, and $P \subseteq V \times (V \cup \Sigma)^*$ is a set of productions. An example of a CFG $\mathcal{G}$ is shown in Fig. 1; its nonterminals V consists of

```
(1) expr PLUS expr PLUS expr
(2) expr STAR expr STAR expr
(3) expr PLUS expr STAR expr
(4) IF expr THEN IF expr THEN stmt ELSE stmt
```

Fig. 2. Expressions demonstrating the ambiguities in the grammar from Fig. 1.

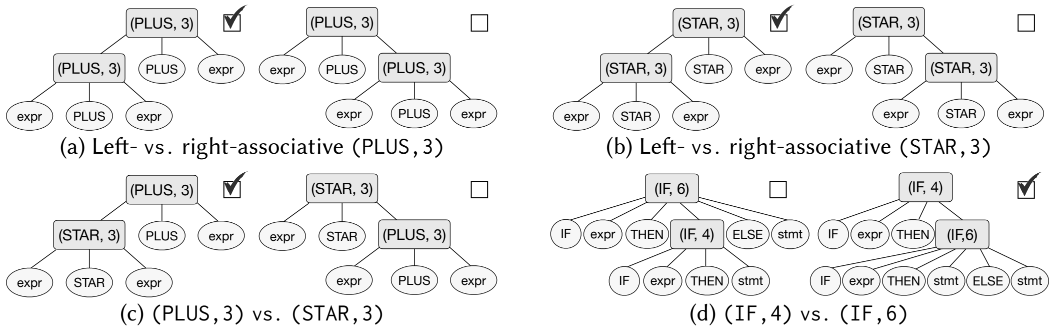

(PLUS, 3)
expr
PLUS
STAR
(STAR, 3)
(IF, 6)
(IF, 4)
IF
THEN
ELSE
stmt
(a) Left- vs. right-associative (PLUS,3)
(b) Left- vs. right-associative (STAR,3)
(c) (PLUS,3) vs. (STAR,3)
(d) (IF,4) vs. (IF,6)

Fig. 3. Tree examples representing conflicts in $\mathcal{G}$.

`stmt`, `decl`, `expr` and `ident`, with a start nonterminal `stmt`; its terminals Σ include `SEMI`, `TINT`, `IDENT`, and `EQ` involved in declaration statements, `IF`, `THEN`, and `ELSE` for conditional statements with only a then-branch or both then- and else-branches, `PLUS`, `STAR`, and `INT` for expressions with binary operations, and `LPAREN` and `RPAREN` for open and close parentheses; its production rules P shown in Fig. 1 replace nonterminal symbols with sequences of nonterminal and/or terminal symbols.

A CFG is called *ambiguous* when there are multiple ways to match a string of terminals via its production rules. $\mathcal{G}$ in Fig. 1 is an example of such grammar, with four different ambiguities, *a.k.a. conflicts*. In particular, these conflicts are caused by lack of information about associativity of `PLUS` and `STAR` operators, and about the precedence order between `PLUS` and `STAR` as well as between `IF` with `THEN` branch and `IF` with `THEN` and `ELSE` branches. Fig. 2 shows expressions that are subject to the conflicts. The first expression `(1)` can be derived by either applying the grammar's production rule `expr` $\rightarrow$ `expr PLUS expr` on the left `expr PLUS expr` first and the right one next *or* the other way around, since the grammar is ambiguous on whether `PLUS` is left- or right-associative. Similarly, the second expression `(2)` is subject to the ambiguity of `STAR` being left-associative or right-associative. The third expression `(3)` is subject to the unspecified precedence orders between `PLUS` and `STAR`. The fourth expression `(4)` illustrates an ambiguity of parsing nested `IF`-expressions with only then- or both then- and else-branches, famously known as the *dangling else* problem [1].

These different ways to parse the same expression can be depicted via *tree examples*, as shown in Fig. 3. We rely on the hierarchical property of such trees: symbols in tree examples that are *deeper* have *higher* precedence orders than symbols at lower depths. These examples are sub-parse trees that represent ambiguities in a CFG and thus use only a subset of the transitions. When only one symbol is present in a tree example, we consider it to represent an associativity ambiguity. Our idea is to allow the user to choose *one such tree per conflict*. The resulting combined set of *tree examples* that are *not* selected by the user T^- can be then used to disambiguate the grammar, updating its rules accordingly. In a nutshell, Greta achieves that by (a) generating a TA from the user-specified tree examples and from the initial (input) CFG and (b) intersecting the learned TA with a TA obtained from the original grammar to resolve the ambiguities (Fig. 4). In the rest of this section, we will walk through the stages of Greta working using the conflict-featuring grammar $\mathcal{G}$ as an example.

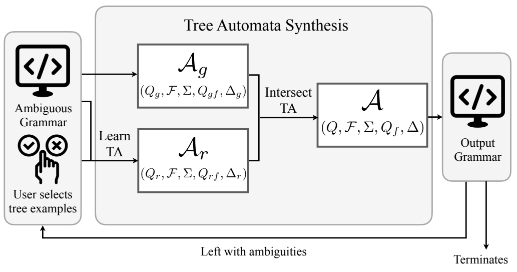

Tree Automata Synthesis
Ambiguous Grammar
User selects tree examples
$\mathcal{A}_g$
$(Q_g, \mathcal{F}, \Sigma, Q_{gf}, \Delta_g)$
Learn TA
$\mathcal{A}_r$
$(Q_r, \mathcal{F}, \Sigma, Q_{rf}, \Delta_r)$
Intersect TA
$\mathcal{A}$
$(Q, \mathcal{F}, \Sigma, Q_f, \Delta)$
Output Grammar
Left with ambiguities
Terminates

Fig. 4. Greta grammar disambiguation workflow.

2.2 From a Context-Free Grammar to a Tree Automaton

The first step in our approach is to convert the input grammar into a tree automaton. It is well-known that any CFG can be represented by a tree automaton that recognises the language of its parse trees—so-called *regular tree language* [10, 13]. We now provide a brief explanation of basic concepts of TAs and show how the grammar from Fig. 1 is translated to its corresponding TA.

A *tree automaton* (TA) is a tuple $(Q, \mathcal{F}, \Sigma, Q_f, \Delta)$ where Q refers to a set of states, $\mathcal{F}$ is a set of constructor labels (*a.k.a.* ranked alphabet), Σ is a set of terminal symbols, $Q_f \subseteq Q$ is a set of final states, and Δ is a set of transition rules. Each ranked symbol in $\mathcal{F}$ has an associated arity (*Rank*) corresponding to the number of terminals and nonterminals that appear on the right-hand side of a production in the original CFG. It also has a symbol (*Sym*) which is unique to each production. A TA accepts a tree if there exists a run from the leaves to a final state at the root, following the transition rules in Δ (described formally in the supplementary material [20]). Fig. 5 shows a readable set of ranked symbols for $\mathcal{G}$, chosen for use in this paper.

A TA $\mathcal{A}_g$ translated from the grammar $\mathcal{G}$ is shown in Fig. 6, combined with the ranked alphabet from Fig. 5. It is generated from the grammar as follows. First, the productions of $\mathcal{G}$ are converted to ranked alphabet symbols with their corresponding arities to result in Fig. 5. A nonterminal-to-nonterminal production like `expr` $\rightarrow$ `ident` is mapped to a transition `expr` $\leftarrow_{(\delta,1)}$ `ident` using a $(\delta, 1)$-label, which is added to the ranked alphabet. The set of nonterminal symbols V and the start symbol S of $\mathcal{G}$ are mapped to a set of states $\mathtt{Q_g}$ and a singleton set of S as the final accepting state $\mathtt{Q_{gf}}$, respectively, as shown in Fig. 6. The set of terminal symbols of $\mathcal{G}$ becomes the set of terminal symbols Σ of the TA. Lastly, production rules of $\mathcal{G}$ are annotated with each rule's ranked alphabet symbol, resulting in transition rules of $\mathcal{A}_g$.

2.3 Generating a Tree Automaton from Tree Examples and an Input Grammar

As we learned from Sec. 2.1 ambiguities in a grammar are caused by the lack of information about associativity and/or precedence orders among terminal symbols. We observe that *semi-concrete* parse trees, as shown in Fig. 3, can be useful for illustrating the options regarding associativity and precedence orders. In our approach, we show such trees to the language designer, asking them to select the options that correspond to their preferred hierarchies among the alphabet symbols in question. Based on the user's selections as well as the original CFG, we generate a TA that rejects unwanted parse trees. The TA must also accept trees consistent with selections and with symbols in $\mathcal{G}$ unrelated to the ambiguities.

2.3.1 Conflict Examples and Their Resolution. Each of the ambiguities coming from the expressions in Fig. 2 is presented to the user as parse trees with different parsing orders, as shown in Fig. 3. Our approach relies on a standard LR(1) parser generator to identify conflicts and generate the tree examples from the sets of productions featuring the ambiguities. For example, the expression `(1) expr PLUS expr PLUS expr` in Fig. 2 can be parsed by applying a production rule `expr` $\rightarrow$ `expr PLUS expr` first and then applying the same rule to the left—or the right—`expr` on the right-hand side of the production, essentially exemplifying the left- or the right-associativity of the `(PLUS,3)` operator. Once the rule is labeled with its respective tree-constructor label, `(PLUS,3)` in this case, the trees representing different ways of parsing are constructed, as shown in Fig. 3a. Likewise, trees representing left- and right-associativity of `(STAR,3)` in Fig. 3b represent two different ways to

```
𝓕 = { (SEMI,2), (IF,4), (IF,6), (TINT,4), (PLUS,3), (STAR,3), (INT,1)
      ((),3), (δ,1), (IDENT,1) }
```

Fig. 5. Ranked alphabet $\mathcal{F}$ for the running example.

```
Qg = { stmt, decl, expr, ident }
Qgf = { stmt }
Δg =
  { stmt ←(SEMI,2)   decl SEMI                      expr←(PLUS,3) expr PLUS expr
    stmt ←(IF,4)     IF expr THEN stmt              expr←(STAR,3) expr STAR expr
    stmt ←(IF,6)     IF expr THEN stmt ELSE stmt    expr←(INT,1)  INT
    decl ←(TINT,4)   TINT ident EQ expr             expr←((),3)   LPAREN expr RPAREN
    ident←(IDENT,1)  IDENT                          expr←(δ,1)    ident              }
```

Fig. 6. TA $\mathcal{A}_g$ reinterpreted from $\mathcal{G}$.

parse (2) `expr STAR expr STAR expr` in Fig. 2. As observed in these sets of productions, if a conflict involves just one symbol, we can infer that it is because of the symbol's unclear associativity.

Other than associativity, the conflicts might be caused by ambiguous precedence orders among different symbols. Consider (3) `expr PLUS expr STAR expr` in Fig. 2. The expression can be parsed by applying the production `expr` → `expr PLUS expr` first and then `expr` → `expr STAR expr` to the right `expr`, generating a parse tree on the left in Fig. 3c. Alternatively, `expr` → `expr STAR expr` can be applied first and then `expr` → `expr PLUS expr` to the left `expr`, generating a tree on the right in Fig. 3c. These represent two alternative ways to generate parse trees by putting these symbols at different *depths* in those trees when precedence orders between these symbols (*e.g.*, `(PLUS,3)` and `(STAR,3)`) are ambiguous. Similarly, ambiguity from parsing the expression (4) `IF expr THEN IF expr THEN stmt ELSE stmt` in Fig. 2, known as the *dangling else problem*, is shown as trees in Fig. 3d, where the left tree symbolises `(IF,4)` having a higher precedence order than `(IF,6)` and the right one with an opposite precedence relation.

Coming back to our running example, suppose the user has indicated their parsing preferences by selecting the ticked trees in Fig. 3. This results in Greta learning the relations of the tree-constructor labels `(PLUS,3)` < `(STAR,3)` and `(IF,4)` < `(IF,6)` in addition to `(PLUS,3)` and `(STAR,3)` being left-associative. These user preferences can be thought of as *restrictions* on the input grammar. Greta collects these restrictions by traversing each of the trees *not* selected by the user to combine them with the so-called *base precedence order* of the original grammar.

The base precedence order O_{bp} denotes the set of all the ranked alphabet symbols obtained from the tree automaton $\mathcal{A}_g$ representing the input CFG $\mathcal{G}$ with their corresponding orders, which can be thought of as the *shortest distance* to the final accepting state of $\mathcal{A}_g$ in Fig. 6. For our example whose ranked alphabet denoted by $\mathcal{F}$ is shown in Fig. 5, O_{bp} is as follows:

```
{((IF,4),0), ((IF,6),0), ((SEMI,2),0), ((TINT,4),1), ((PLUS,3),1), ((STAR,3),1),
 ((INT,1),1), (((),3),1), ((δ,1),1)}
```

The 1-arity symbol `IDENT` is identified as a *trivial symbol* (detailed in section 3.1.1) and is handled separately, since we know that it will not be conflicting with other symbols, and their corresponding nonterminals remain intact throughout the run of Greta. Its associated state `ident` as well as transition `ident` $\leftarrow_{(\texttt{IDENT},1)}$ `IDENT` remain unchanged in the generated TA. We provide a formal definition of O_{bp} and describe a procedure to construct it in Sec. 3.1.

2.3.2 Parsing Preferences as a Tree Automaton. The following step—constructing a tree automaton that elaborates the input grammar with additional restrictions—is the key novel idea of this work.

Notice that the trees in Fig. 3 are *not* parse trees of the input grammar $\mathcal{G}$; rather they are *sub-trees* of some of the parsing trees allowed by the grammar. To wit, they feature symbols `(PLUS,3)`, `(STAR,3)`, `(IF,4)`, and `(IF,6)`, while missing information about precedence orders of other symbols from the alphabet $\mathcal{F}$, such as `(SEMI,2)` or `(TINT,4)` with respect to symbols in the examples, as that information can be restored via O_{bp}. Given the preferences indicated by the user on the provided

```
Qr = { e0, e1, e2, e3, e4, ident }
Qrf = { e0 }
Δr =
   { e0  ←(IF,4)    IF e0 THEN e0              e3  ←(STAR,3) e3 STAR e4
     e0  ←(SEMI,2)  e0 SEMI                    e3  ←(TINT,4) TINT ident EQ e3
     e0  ←(ϵ,1)     e1                         e3  ←(INT,1)  INT
     e1  ←(IF,6)    IF e1 THEN e1 ELSE e1      e3  ←((),3)   LPAREN e3 RPAREN
     e1  ←(SEMI,2)  e1 SEMI                    e3  ←(δ,1)    ident
     e1  ←(ϵ,1)     e2                         e3  ←(ϵ,1)    e4
     e2  ←(PLUS,3)  e2 PLUS e3                 e4  ←(TINT,4) TINT ident EQ e4
     e2  ←(TINT,4)  TINT e2 EQ e2              e4  ←(INT,1)  INT
     e2  ←(INT,1)   INT                        e4  ←((),3)   LPAREN e4 RPAREN
     e2  ←((),3)    LPAREN e2 RPAREN           e4  ←(δ,1)    ident
     e2  ←(δ,1)     ident                      ident ←(IDENT,1) IDENT
     e2  ←(ϵ,1)     e3                                                       }
```

Fig. 7. The tree automaton $\mathcal{A}_\mathrm{r}$ generated from user-specified parsing preferences.

examples, our goal is to capture them into a *new* tree automaton $\mathcal{A}_\mathrm{r}$ that is also "permissive enough" to accept the desired "complete" parse trees allowed by the TA $\mathcal{A}_\mathrm{g}$ corresponding to the original grammar. In the remainder of this section, we provide an informal description of this procedure, fleshing out its details and correctness argument in Sec. 3.1.

We start by combining the restrictions learned from the examples in Fig. 3 (*e.g.*, `(PLUS,3)` < `(STAR,3)`, `(IF,4)` < `(IF,6)`, *etc*) with the base precedence order O_bp, obtaining the following set of precedence orders—let's call it O_p—for all symbols:

```
{ ((IF,4),0), ((SEMI,2),0), ((IF,6),1), ((SEMI,2),1), ((PLUS,3),2), ((TINT,4),2),
  ((INT,1),2), (((),3),2), ((δ,1),2), ((STAR,3),3), ((TINT,4),3), ((INT,1),3),
  (((),3),3), ((δ,1),3), ((TINT,4),4), ((INT,1),4), (((),3),4), ((δ,1),4) }
```

When orders of symbols from the chosen examples are compared in this updated set O_p, it is consistent with the precedence relations `(PLUS,3)` < `(STAR,3)` as well as `(IF,4)` < `(IF,6)`. In addition, for all other pairs of symbols in the base precedence order, there exists a pair in O_p for those symbols with the same relative order. O_p also accommodates for later integration with associativity restrictions O_a, full details of which are explained in Sec. 3.1.

Subsequently, given a pair of ranked symbol and its order (f, o) in O_p, Greta generates an f-labeled transition from the states and terminals in the right-hand side of *Prod*(f) to $\mathtt{e}_o$. As an example, `((IF,4),0)` from O_p produces the transition $\mathtt{e}_0 \leftarrow_{(\mathtt{IF},4)}$ `IF` $\mathtt{e}_0$ `THEN` $\mathtt{e}_0$. At the same time, each ordered state $\mathtt{e}_o$ is linked to its next ordered state in hierarchy $\mathtt{e}_{o+1}$ by an ϵ-transition: *e.g.*, $\mathtt{e}_0 \leftarrow_{(\epsilon,1)} \mathtt{e}_1$. The state $\mathtt{e}_0$ corresponding to the symbol `(IF,4)` represents a level in the hierarchy of parsing orders. Moreover, if a symbol was used to specify an associativity in the examples, Greta generates a transition that takes it into account: *e.g.*, $\mathtt{e}_2 \leftarrow_{(\mathtt{PLUS},3)} \mathtt{e}_2$ `PLUS` $\mathtt{e}_3$ produced based on left-associative `(PLUS,3)` from O_a and `((PLUS,3),2)` from O_p. These steps lead to construction of $\mathcal{A}_\mathrm{r}$ in Fig. 7. The details of the algorithm are explained in Sec. 3.1.

Notice that the automaton obtained this way is not immediately the result we want that corresponds to the repaired grammar. That is because the constructed $\mathcal{A}_\mathrm{r}$ recognises not only all those non examples-related parse trees allowed by $\mathcal{A}_\mathrm{g}$ but *even* those terms that are not allowed by $\mathcal{A}_\mathrm{g}$. For example, consider an expression `IF 1 THEN 2 STAR 3`. A tree corresponding to this expression is accepted by $\mathcal{A}_\mathrm{r}$ since $\mathtt{e}_1$ appearing after `THEN` can be rewritten by $\mathtt{e}_3$ `STAR` $\mathtt{e}_4$, whereas this is not possible in the original grammar, hence *not* accepted by $\mathcal{A}_\mathrm{g}$. This is the reason why we need to

```
(stmt,e0) ←(SEMI,2) (decl,e0) SEMI
(stmt,e0) ←(SEMI,2) (decl,e1) SEMI
(stmt,e0) ←(IF,4) IF (expr,e0) THEN (stmt,e0)
(stmt,e0) ←(IF,6) IF (expr,e1) THEN (stmt,e1) ELSE (stmt,e1)

(decl,e0) ←(TINT,4) TINT (ident,ident) EQ (expr,e2)
(decl,e0) ←(TINT,4) TINT (ident,ident) EQ (expr,e3)
(decl,e0) ←(TINT,4) TINT (ident,ident) EQ (expr,e4)

(decl,e1) ←(TINT,4) TINT (ident,ident) EQ (expr,e2)
(decl,e1) ←(TINT,4) TINT (ident,ident) EQ (expr,e3)
(decl,e1) ←(TINT,4) TINT (ident,ident) EQ (expr,e4)

(expr,e0) ←(PLUS,3) (expr,e2) PLUS (expr,e3)
(expr,e0) ←(INT,1) INT
(expr,e0) ←(δ,1) (ident,ident)
(expr,e0) ←((),3) LPAREN (expr,e2) RPAREN
(expr,e0) ←((),3) LPAREN (expr,e3) RPAREN
(expr,e0) ←((),3) LPAREN (expr,e4) RPAREN

(expr,e1) ←(PLUS,3) (expr,e2) PLUS (expr,e3)
(expr,e1) ←(INT,1) INT
(expr,e1) ←(δ,1) (ident,ident)
(expr,e1) ←((),3) LPAREN (expr,e2) RPAREN
(expr,e1) ←((),3) LPAREN (expr,e3) RPAREN
(expr,e1) ←((),3) LPAREN (expr,e4) RPAREN

(stmt,e1) ←(SEMI,2) (decl,e1) SEMI
(stmt,e1) ←(IF,6) IF (expr,e1) THEN (stmt,e1) ELSE (stmt,e1)

(ident,ident)←(IDENT,1) IDENT

(expr,e2) ←(PLUS,3) (expr,e2) PLUS (expr,e3)
(expr,e2) ←(INT,1) INT
(expr,e2) ←(δ,1) (ident,ident)
(expr,e2) ←((),3) LPAREN (expr,e2) RPAREN
(expr,e2) ←((),3) LPAREN (expr,e3) RPAREN
(expr,e2) ←((),3) LPAREN (expr,e4) RPAREN

(expr,e3) ←(STAR,3) (expr,e3) STAR (expr,e4)
(expr,e3) ←(INT,1) INT
(expr,e3) ←(δ,1) (ident,ident)
(expr,e3) ←((),3) LPAREN (expr,e3) RPAREN
(expr,e3) ←((),3) LPAREN (expr,e4) RPAREN

(expr,e4) ←(INT,1) INT
(expr,e4) ←(δ,1) (ident,ident)
(expr,e4) ←((),3) LPAREN (expr,e4) RPAREN
```

(a) Cross product result of Δ_g and Δ_r.

```
stmt0 ←(SEMI,2) decl SEMI
stmt0 ←(SEMI,2) decl SEMI
stmt0 ←(IF,4) IF expr0 THEN stmt0
stmt0 ←(IF,6) IF expr1 THEN stmt1 ELSE stmt1

decl ←(TINT,4) TINT ident EQ expr0
decl ←(TINT,4) TINT ident EQ expr1
decl ←(TINT,4) TINT ident EQ expr2

stmt1 ←(SEMI,2) decl SEMI
stmt1 ←(IF,6) IF expr1 THEN stmt1 ELSE stmt1

expr0 ←(PLUS,3) expr2 PLUS expr1
expr0 ←((),3) LPAREN expr0 RPAREN
expr0 ←((),3) LPAREN expr1 RPAREN
expr0 ←(INT,1) INT
expr0 ←(δ,1) ident
expr0 ←((),3) LPAREN expr2 RPAREN

expr1 ←(STAR,3) expr1 STAR expr2
expr1 ←((),3) LPAREN expr1 RPAREN
expr1 ←(INT,1) INT
expr1 ←(δ,1) ident
expr1 ←((),3) LPAREN expr2 RPAREN

expr2 ←(INT,1) INT
expr2 ←(δ,1) ident
expr2 ←((),3) LPAREN expr2 RPAREN

ident ←(IDENT,1) IDENT
```

(b) After removing duplicates and renaming states.

Fig. 8. Minimisation of the cross product of transitions Δ_g and Δ_r.

take an intersection of the tree automata to eventually produce a grammar that is both *not* too permissive *wrt.* original grammar and restrictive *wrt.* user-specified parsing preferences.

2.4 Repairing the Grammar by Intersecting Tree Automata

We now have two tree automata: $\mathcal{A}_g \triangleq (Q_g, \mathcal{F}, \Sigma, Q_{gf}, \Delta_g)$ from $\mathcal{G}$ and $\mathcal{A}_r \triangleq (Q_r, \mathcal{F}, \Sigma, Q_{rf}, \Delta_r)$ learned from the tree examples as well as $\mathcal{G}$. The outcome of our example-based ambiguity repair is captured by the tree automaton $\mathcal{A}_{res} \triangleq (Q, \mathcal{F}, \Sigma, Q_f, \Delta)$ obtained as an intersection of $\mathcal{A}_g$ and

$\mathcal{A}_r$. It is defined as a tuple that consists of cross-products of each component [10]: *i.e.*, $Q = Q_g \times Q_r$, $Q_f = Q_{gf} \times Q_{rf}$, and $\Delta = \Delta_g \times \Delta_r$. Let us discuss each component for our example.

First, taking the product of the accepting states of $\mathcal{A}_g$ and $\mathcal{A}_r$ produces the new state `(stmt,e`$_0$`)`, which is the start for the intersected TA. Next, starting from the state `(stmt,e`$_0$`)` as the left-hand side of the transition, going across `stmt`-producing transitions in Δ_g and `e`$_0$-producing transitions in Δ_r reveals that there are three constructor labels —`(SEMI,2)`, `(IF,4)`, and `(IF,6)`—with all the *matching* right-hand sides. Note `x`-producing transitions essentially refer to those rules that transition to the state `x` in the TA. Given a state (α_g, α_r), the transitions $\alpha'_g \leftarrow \beta_g \in \Delta_g$ and $\alpha'_r \leftarrow \beta_r \in \Delta_r$ (where α'_g and α'_r are α_g and α_r respectively, or reachable from them by epsilon transitions) are considered *matching* when the lengths (number of terminals and states) in β_g and in β_r are identical *and* at each position, either both elements are states or are *the same* terminal.

For example, consider the transition for symbol `(IF,4)`. In Δ_g, the transition `stmt` $\leftarrow_{\texttt{(IF,4)}}$ `IF expr THEN stmt`, matches transition `e`$_1$ $\leftarrow_{\texttt{(IF,4)}}$ `IF e`$_1$`THEN e`$_1$ in Δ_r because their right-hand sides have the same number of elements *and* at each position, it is either both states or the same terminal symbol. On the other hand, when we look at the symbol `(PLUS,3)`, we cannot find a corresponding transition in Δ_g producing `stmt`, taking `(PLUS,3)` as a constructor label. This results in no `(stmt,e`$_1$`)`-producing transition for the symbol `(PLUS,3)` in the tree automata intersection.

Following this intuition, taking cross-product of transitions producing `(stmt, e`$_0$`)` results in transitions in light grey with labels `(SEMI,2)`, `(IF,4)`, `(IF,6)` in Fig. 8a. Moreover, it shows that there are following states that can be *reached* from the `(stmt,e`$_0$`)`-producing transitions: `(decl,e`$_0$`)`, `(decl,e`$_1$`)`, `(expr,e`$_0$`)`, `(expr,e`$_1$`)`, `(stmt,e`$_0$`)` and `(stmt,e`$_1$`)`. We look at these so-called *reachable* states, identifying transitions that produce each of them, so that considering those transitions might add new states to the reachable states. This generates the rest of the transition rules in Fig. 8a.

Once all the transitions producing the reachable states are obtained, these rules are examined in order to get rid of any duplicate states and accordingly the rules that transition to the duplicate states. In our running example, the states `(expr, e`$_0$`)`, `(expr,e`$_1$`)` and `(expr,e`$_2$`)` as well as `(decl, e`$_0$`)` and `(decl, e`$_1$`)` are identified as duplicates. We highlight the former sets of transitions in darker grey, which all have identical right-hand sides. Hence, we remove two of them—let's say, `(expr,e`$_1$`)` and `(expr,e`$_2$`)`—and their transition rules, while also replacing all their occurrences with `(expr,e`$_0$`)`. We do this also for `(decl,e`$_0$`)` and `(decl,e`$_1$`)`. After removing these duplicates and renaming the states, we obtain a smaller, yet equivalent, set of transitions in Fig. 8b.

Lastly, we introduce $(\epsilon, 1)$-transitions to further simplify the resulting transitions. For example, looking at the `expr`$_2$-producing transitions, `expr`$_0$- and `expr`$_1$-producing transitions repeat them, as indicated by the transitions in dotted boxes in Fig. 8b. That is, the transition rules take the same set of tree-constructor labels and respectively transition to the same set of right-hand sides. Hence, we can simplify `expr`$_0$- and `expr`$_1$-producing transitions by respectively adding the rules `expr`$_0$$\leftarrow_{(\epsilon,1)}$ `expr`$_2$ and `expr`$_1$$\leftarrow_{(\epsilon,1)}$ `expr`$_2$, while removing the repeated transitions. We repeat this process for all the states and their transitions, resulting in a final TA $\mathcal{A}_{res}$ in Fig. 9. The obtained tree automaton $\mathcal{A}_{res}$ (Fig. 9) is converted back to its corresponding CFG trivially, by un-labeling the transitions.[1] This grammar is then passed back to Greta. If there are any remaining ambiguities in it, then Greta performs all the previous steps again. It stops running when there are no more ambiguities in the grammar, *or* any remaining ones are outside the scope of Greta or Menhir, details of which we discuss in Sec. 4.4.2. Greta eventually terminates because the original CFG and tree automata constructed from it as well as its ambiguities are finite, while the number of

[1]Since ϵ-transitions don't consume tree nodes (represented with symbols), they *cannot* be trivially unlabelled, as they lead to productions in trees that weren't in the original grammar. Instead, the unit productions after unlabelling need to be removed, to produce a new CFG. In Menhir, we circumvent this with semantic actions in curly braces, (*e.g.* `expr0 -> expr1 { $1 }`) with which we can avoid having such productions appear in the final AST.

```
Q = { stmt0, stmt1, expr0, expr1, expr2, ident }
Qf = { stmt0 }
Δ =
  { stmt0 ←(IF,4) IF expr0 THEN stmt0                   expr0 ←(PLUS,3) expr0 PLUS expr1
    stmt0 ←(ϵ,1) stmt1                                  expr0 ←(ϵ,1) expr1
    stmt1 ←(SEMI,2) decl SEMI                           expr1 ←(STAR,3) expr1 STAR expr2
    stmt1 ←(IF,6) IF expr0 THEN stmt1 ELSE stmt1        expr1 ←(ϵ,1) expr2
    decl ←(TINT,4) TINT ident EQ expr0                  expr2 ←(INT,1) INT
    ident ←(IDENT,1) IDENT                              expr2 ←(ϵ,1) ident
                                                        expr2 ←((),3) LPAREN expr0 RPAREN }
```

Fig. 9. An automaton $\mathcal{A}_{\text{res}}$ resulted from intersection of $\mathcal{A}_{\text{g}}$ and $\mathcal{A}_{\text{r}}$.

ambiguities (and hence the number of parse trees admitted by the CFG) decreases with each run of Greta (Sec. 4.4.2). We present our intersection algorithm in Sec. 3.2.

2.5 Putting It All Together

The overall workflow of grammar disambiguation in Greta framework works as follows, as illustrated in Fig. 4. If an ambiguous CFG is provided as an input, it is interpreted as a tree automaton, and the ambiguities in it are presented to the user, asking the user, for each of them, to select one of the two alternative tree examples, which together represent different ways to parse the same expression per conflict (Sec. 2.3.1). Once the user specifies their preferences by selecting a set of the tree examples, Greta subsequently uses the chosen examples as well as the precedence orders of all the alphabet symbols from the initial grammar to learn a TA encoding the user's parsing preferences in line with the grammar (Sec. 2.3.2). Next, the two TAs are intersected to result in a new TA (Sec. 2.4), which is then translated back to its corresponding grammar. If the resulting grammar is still left with any ambiguities, Greta is run on it again until the grammar is fully disambiguated or only left with non-addressable ones. In this process, the user is simply involved in clarifying parsing preferences in the form of tree examples, without having to compute TA operations or even knowing that these operations are done at all.

3 Grammar Repair by Example, Formally

This section describes technical details of the Greta framework. First, we present an algorithm for learning a TA from tree examples and a base grammar, with its soundness guarantees in Sec. 3.1. Next, we explain the algorithm we use for intersecting TAs and show its correctness Sec. 3.2.

3.1 From Tree Examples to a Tree Automaton

The learning process can be largely divided into two parts: (a) learning restrictions about the associativity and precedence order through LearnOaOp (Algorithm 3.1) from the tree examples and (b) subsequently constructing a TA via GenTA procedure (Algorithm 3.2). Before we describe each of the algorithms, we provide a formal description of tree examples and tree automata.

Unlike a complete parse tree, tree examples can start from any nonterminal, and can have nonterminals at the leaves instead of terminals, *e.g.* $\mathsf{PLUS}_3(\mathsf{STAR}_3(\mathsf{expr}, *, \mathsf{expr}), +, \mathsf{expr})$. An example tree represents a set of valid complete parse trees for which (1) all child nonterminals in the tree example are substituted with valid subtrees rooted at that nonterminal, and (2) the resulting tree after substitution is a subtree of the complete CFG parse tree. Given a tree example t, we write $\mathit{ParseTrees}(t)$ to denote the set of parse trees it represents. Suppose we have tree languages $\mathcal{L}_{\text{g}}$

representing the set of parse trees of the grammar $\mathcal{G}$ rooted at the start nonterminal, $\mathcal{T}_{\text{g}}^{nt}$ representing parse trees rooted at the nonterminal nt in $\mathcal{G}$, and functions $\mathcal{T}_{\text{g}}^{nt} \to \mathcal{L}_{\text{g}}$ representing parse tree contexts for some nonterminal nt in $\mathcal{G}$. Then, tree examples with k incomplete nonterminals $nt_1, \ldots, nt_k$, rooted at nt are functions $\mathcal{T}_{\text{g}}^{nt_1} \times \cdots \times \mathcal{T}_{\text{g}}^{nt_k} \to \mathcal{T}_{\text{g}}^{nt}$, and $\mathit{ParseTrees}(t)$ is defined as:

$$\begin{aligned}\mathit{ParseTrees}(t) \triangleq \{t' \mid \exists\, nt, nt_1, \ldots, nt_k, \exists\, C \in \mathcal{T}_{\text{g}}^{nt} \to \mathcal{L}_{\text{g}}, t \in \mathcal{T}_{\text{g}}^{nt_1} \times \cdots \times \mathcal{T}_{\text{g}}^{nt_k} \to \mathcal{T}_{\text{g}}^{nt},\\ \exists\, t_1 \in \mathcal{T}_{\text{g}}^{nt_1}, \ldots, t_k \in \mathcal{T}_{\text{g}}^{nt_k}.\ t' = C[t(t_1, \ldots, t_k)]\}\end{aligned}$$

Greta currently supports tree examples that use exactly two productions, which covers the vast majority of ambiguities in practice. We define some convenient notation to work with these examples: given a tree example t, t_T denotes the root (top) symbol of t, t_B denotes the nested (bottom) symbol, t_{idx} denotes the index i (starting from 0 on the left) of the child node at which t_B appears, where $0 \le i < \mathit{Rank}(t_T)$. Given symbols α, β and index i, $\mathit{Eg}(\alpha, \beta, i)$ denotes the tree example such that $\mathit{Eg}(\alpha, \beta, i)_T = \alpha$, $\mathit{Eg}(\alpha, \beta, i)_B = \beta$, $\mathit{Eg}(\alpha, \beta, i)_{idx} = i$ and all other child nodes are wildcards. For example, $\mathit{Eg}(\texttt{PLUS}_3, \texttt{STAR}_3, 0)$ denotes the tree example $\texttt{PLUS}_3(\texttt{STAR}_3(\texttt{expr}, *, \texttt{expr}), +, \texttt{expr})$. An alternative to a tree example will involve the same two symbols in a different configuration. The tree examples supported by Greta are the ones that can be expressed with Eg.

An example t is an associativity-related example if $t_T = t_B$, and a precedence order-related example if $t_T \ne t_B$. When a user is presented with two tree examples, the one that is not selected, corresponds to trees that we want to exclude from the repaired grammar. $P^-(t)$ denotes the set of parse trees to remove, corresponding to a tree example t not selected by the user:

$$P^-(t) \triangleq \begin{cases} \mathit{ParseTrees}(t) & \text{if } t_T = t_B \\ \bigcup\limits_{0 \le i < \mathit{Rank}(t_T)} \mathit{ParseTrees}(\mathit{Eg}(t_T, t_B, i)) & \text{otherwise} \end{cases}$$

In resolving precedence order related conflicts, we enforce a hierarchy of symbols in which a symbol of higher precedence appears strictly deeper in the tree than a symbol of lower precedence, hence requiring a union of excluded parse trees for all possible child positions. From the set of symbols $\mathcal{S}_{\text{C}}$ involved in precedence or associativity related conflicts, Greta constructs the following partition of $\mathcal{S}_{\text{C}}$:

$$\begin{aligned}\mathcal{S}_{\text{E}} = \{S \subseteq \mathcal{S}_{\text{C}} \mid \forall s_i, s_j \in S, s_i \ne s_j, \exists\, n, m \in \mathbb{N} \text{ such that}\\ \mathit{ParseTrees}(\mathit{Eg}(s_i, s_j, n)) \ne \emptyset \lor \mathit{ParseTrees}(\mathit{Eg}(s_j, s_i, m)) \ne \emptyset,\ S \text{ is maximal}\}\end{aligned}$$

Each set of symbols in $\mathcal{S}_{\text{E}}$ are the maximal subsets of $\mathcal{S}_{\text{C}}$ such that for all pairs s_i and s_j in the set, the grammar (describing language $\mathcal{L}_{\text{g}}$) permits trees $\mathit{Eg}(s_i, s_j, n)$ *or* $\mathit{Eg}(s_j, s_i, m)$ for some child positions n and m.

For each of these sets, Greta generates precedence-related tree examples for the pairs $\mathit{Eg}(s_i, s_j, n)$ and $\mathit{Eg}(s_j, s_i, m)$ if they can be parsed in both orders, to obtain a total order of the symbols in the set. It also generates associativity-related tree examples for those in conflict. It then understands the desired restrictions by interaction with the user.

We have now laid out notions of tree examples, associativity, precedence, and excluded parse trees. Our formalism for tree automata and conversion from CFGs to TAs is relatively standard, and can be found in the supplementary material [20]. We now turn to Greta's TA learning algorithm.

3.1.1 *Base Precedence Order.*

Algorithm 3.1 starts by computing a set of existing precedence orders for all the symbols in $\mathcal{F}$ in $\mathcal{G}$, referred to as the *base precedence order* O_{bp}, in the following way:

- First, a *level* (or the distance from the start nonterminal) for each nonterminal $e \in V$ is
  - $d(e) = 0$ if e is a start nonterminal of $\mathcal{G}$.

  - Otherwise, $d(e)$ is the smallest n such that there exists a sequence of nonterminals $nt_0, \ldots, nt_n$ where $nt_n = e$, nt_0 is the start nonterminal, and for each nt_i, nt_{i+1}, there exists a production $nt_i \longrightarrow \beta$ in P such that β contains nt_{i+1}.
- Next, an *order* of each symbol s in $\mathcal{F}$ is determined using the order function $\widehat{o}\colon \mathcal{F} \mapsto \mathbb{Z}$, defined $\widehat{o}(s) \triangleq d(Lhs(Prod(s)))$, where $Prod(s)$ is the unique nonterminal of a ranked symbol, and Lhs denotes the left-hand side nonterminal of a production.
- Lastly, O_{bp} is constructed as a set $\{(s, \widehat{o}(s)) \mid s \in \mathcal{F} \setminus \mathcal{F}_{\text{tr}}\}$, where $\mathcal{F}_{\text{tr}}$ is the set of trivial symbols defined below.

One can think of an order of a symbol s as the shortest distance to reach s from the start nonterminal. In the absence of cycles, a symbol at a higher order always appears deeper in the parse tree than a symbol at a lower order. Correctly stratifying symbol order based on user preference allows us to correct precedence ambiguities between different symbols and associativity ambiguities between a symbol and itself. To account for cycles in order, we will need to reintroduce these cycles later in the TA learning process described in Sec. 3.1.3. Now, looking at the running example $\mathcal{G}$ from Fig. 1 with its start nonterminal `stmt`, we have the following levels for the nonterminals: 0 for `stmt`, 1 for both `decl` and `expr`, and 2 for `ident`. Based on this, we can determine orders for all the symbols. Looking at the symbol `(IF,6)` and its production `stmt` $\longrightarrow$`IF expr THEN stmt ELSE stmt`, for instance, the order of `(IF,6)` is computed d(`stmt`) = 0. Similarly, we can compute orders for all the symbols, producing O_{bp} in Sec. 2.3.

The set of trivial symbols $\mathcal{F}_{\text{tr}}$ is a set of 1-arity symbols defined as follows:

$$\mathcal{F}_{\text{tr}} \triangleq \left\{ s \;\middle|\; \begin{array}{l} s \in \mathcal{F} \wedge Rank(s) = 1 \wedge \delta_s \triangleq \alpha \leftarrow_s \beta \in \Delta \text{ where } \beta \in \Sigma \wedge \\ \forall\, \delta_{s'} \in \Delta \; s.t.\; \delta_{s'} = \alpha \leftarrow_{s'} \beta', \beta' \in \Sigma \end{array} \right\}$$

$\mathcal{F}_{\text{tr}}$ are symbols whose left hand side nonterminals only lead to a single terminal symbol. Because we know that symbols in $\mathcal{F}_{\text{tr}}$ will not be involved in any ambiguities *wrt.* associativity or precedence order, O_{bp} does not include symbols in $\mathcal{F}_{\text{tr}}$ and the productions involving trivial symbols remain intact in the generated TA. Note however, that this separate handling of trivial symbols is only an optimisation and does not affect the correctness of the learning algorithm. It is also the only such optimisation in Algorithm 3.1 and Algorithm 3.2.

The construction of the base order O_{bp} is crucial for obtaining the relative precedence orders of symbols not involved in the tree examples. If a TA were to be synthesised based on tree examples alone, the learned TA would feature only those symbols associated with the ambiguities, which is typically a relatively small subset of the original symbol alphabet. At the same time, we believe that generating examples that would collectively contain all symbols of the input grammar would be detrimental for the usability of our tool. This is why the examples provided by Greta only contains a subset of the grammar's symbols, leaving aside the ones that were not involved in the ambiguities from the original grammar. Preserving the precedence orders O_{bp} of the original grammar allows Greta to learn a TA involving *all* precedence and associativity conflicts efficiently, without compromising soundness *wrt.* its correctness statement in Sec. 3.1.4.

3.1.2 Computing Associativity and Precedence Order. As illustrated in Algorithm 3.1, the LearnOaOp procedure takes tree examples *not* chosen by the user T^-, the input CFG $\mathcal{G}$ and a map M_{to} from order to a set of ordered sets of symbols from $\mathcal{S}_{\text{E}}$ that are at that order, from lowest to highest precedence, inferred from T^-. [2] Intuitively, we care about negative tree examples in particular because the goal of the learning algorithm is to *exclude* undesirable patterns appearing anywhere

[2]Since all pairs of symbols in the sets of $\mathcal{S}_{\text{E}}$ can be in precedence conflict (*i.e.* parsed in either order), they must have the same order (allowing us to use it to index M_{to}) or are at adjacent orders. If they are at adjacent orders, it is sound to merge the symbols at the two orders into one.

ALGORITHM 3.1: LearnOaOp: learning associativity and precedence orderings.

```
Input: tree examples $T^-$, order to ordered symbols map $M_{\text{to}}$, input CFG $\mathcal{G}$
Output: associativity $O_a$, precedence order $O_p$
$O_{bp} \leftarrow$ base precedence order of $\mathcal{G}$
$O_a, O_p \leftarrow \{\}$; $O_{tmp} \leftarrow O_{bp}$ // temporary set initialised with all elements in $O_{bp}$
for $t \in T^-$ do
    if $t_T = t_B$ then
        $O_a \leftarrow O_a \cup \{(t_T, t_{idx})\}$
for descending $(o, G)$ in $M_{to}$ do
    size $\leftarrow$ maxSize$(G)$
    $S = \text{symbols}(\text{ofOrder}(O_{tmp}, o)) \setminus \bigcup_{g \in G} g$
    $O_{tmp} \leftarrow \text{pushN}(O_{tmp}, o+1, size-1) \setminus \text{ofOrder}(O_{tmp}, o)$
    for $i$ in $[0, size)$ do
        ithSymbols $\leftarrow$ getAtIndex$(G, i)$
        $O_{tmp} \leftarrow O_{tmp} \cup \text{withOrder}(S \cup \text{ithSymbols}, o+i)$
        if $i = size - 1 \wedge \exists\, s \in ithSymbols, (s, \_) \in O_a$ then
            $O_{tmp} \leftarrow \text{pushN}(O_{tmp}, o+i+1, 1)$
            $O_{tmp} \leftarrow O_{tmp} \cup \text{withOrder}(S, o+i+1)$
$O_p \leftarrow O_{tmp}$
return $(O_a, O_p)$
```

in the parse tree, as opposed to preserving desired patterns. This is made clear in the description of the algorithm's correctness in Sec. 3.1.4. The algorithm then returns restrictions on associativity O_a and precedence order O_p.

The goal of Algorithm 3.1 is to deduce a set O_a of associativity restrictions (illegal child positions) and a set O_p which enforces a hierarchy of precedence orders among conflicting symbols, while preserving other pairwise orders from O_{bp}. In section 3.1.3, each order in O_p will correspond to states in the learned TA, where some state e_i corresponding to order i can be reached by epsilon transitions from state e_j for $j > i$. With O_{bp}, LearnOaOp collects restrictions related to associativity and precedence order, respectively in O_a and O_p.

First, each tree is examined to check if it is an associativity or precedence order-related example. If it is an associativity-related example, then we store in O_a a pair of the symbol and the position of the nested symbol we want to disallow. For example, in Fig. 3a, the tree *not* selected contains symbol `(PLUS,3)` as a right child (position 1). Then a pair `((PLUS,3), 1)` is added to the set O_a. Combined with the tree selected in Fig. 3b, the user interaction produces a set $O_a = \{$`((PLUS,3), 1)`, `((STAR,3), 1)`$\}$.

Next, the algorithm iterates over M_{to} in descending order of the symbol orders. For each order o and corresponding set of ordered sets of symbols G, it first computes the size of the largest set in G. It then stores in S the set of symbols at order o that are not involved in any conflict. Then, it removes the symbols at order o from O_{tmp} and calls `pushN`, which increments the orders of all symbols of order $\geq o+1$ by $size - 1$, to make room for the new symbols.

Then, for each order $o+i$ from o to $o + size - 1$, it inserts the i-th set of symbols in G, and the auxiliary set of symbols S that are not involved in conflicts. `getAtIndex` returns the symbols at index i in every set in G, if it exists, and `withOrder` assigns the specified order to the provided symbols.

As an example, a map $M_{\text{to}} = \{0 \longrightarrow$ `((STAR,3), (PLUS,3))`$\}$ would update O_{tmp} as follows:

```
{ ((SEMI,2),0), ((PLUS,3),0), ((STAR,3),0) }
                      ↓
{ ((SEMI,2),0), ((STAR,3),0), ((SEMI,2),1), ((PLUS,3),1) }
```

Here, the set $S = \{$`(SEMI,2)`$\}$, `getAtIndex`$(G, 0)$ is `(STAR,3)` and `getAtIndex`$(G, 1)$ is `(PLUS,3)`.

This procedure preserves precedence order relations between pairs of symbols that are not involved in the tree examples. Finally, for all associativity-related symbols in O_a, we want that

ALGORITHM 3.2: GenTA: generating a TA from associativity, precedence, and the input grammar.

```
Input: $O_a$ associativity set, $O_p$ precedence order set, $\mathcal{G} = (V, \Sigma, S, P)$ input CFG
Output: $\mathcal{A} = (Q, \mathcal{F}, \Sigma, Q_f, \Delta)$ a tree automaton
$m \leftarrow$ max order of $O_p$; $Q \leftarrow \{e_0, \ldots, e_m\}$; $Q_f \leftarrow \{e_0\}$
$\mathcal{F} \leftarrow$ ranked symbols of $\mathcal{G}$; $\mathcal{F}_{tr} \leftarrow$ trivial symbols of $\mathcal{G}$
$\delta_{\mathcal{F}} \leftarrow \delta$-generator $w.r.t.$productions $P$
/* Transitions for non-trivial symbols */
for $(s, i) \in O_p$ do
    if $\exists p, (s, p) \in O_a$ then
        // $\delta_s \triangleq e_i \leftarrow_s \ldots e_{i+1} \ldots$ ($e_{i+1}$ at position $p$ in the RHS, with $n$ leading and $m$ trailing
           nonterminals respectively)
        $\Delta \leftarrow \Delta \cup \{\delta_{\mathcal{F}}(e_i, [e_i; \ldots e_{i+1}; \ldots e_i], s)\}$ // $n$ leading and $m$ trailing $e_i$'s
    else
        $\Delta \leftarrow \Delta \cup \{\delta_{\mathcal{F}}(e_i, \overline{e_i}, s)\}$
/* Transitions for trivial symbols */
for $s' \in \mathcal{F}_{tr}$ do
    $Q \leftarrow Q \cup \{e_{s'}\}$; $\Delta \leftarrow \Delta \cup \{\delta_{\mathcal{F}}(e_{s'}, [\,], s')\}$ // i.e., $\delta_{s'} \triangleq e_{s'} \leftarrow_{s'} \alpha$
/* Transitions for connecting ordered states */
for $i \in [0..m-1]$ do
    $\Delta \leftarrow \Delta \cup \{\delta_{\mathcal{F}}(e_i, [e_{i+1}], (\epsilon, 1))\}$ // i.e., $\delta_{(\epsilon,1)} \triangleq e_i \leftarrow_{(\epsilon,1)} e_{i+1}$
/* Handle cycles in order */
for $(sl, ol), (sh, oh) \in HighToLow(G, O_p)$ do
    $\Delta \leftarrow \Delta \cup \{\delta_{\mathcal{F}}(e_{oh}, \overline{e_{ol}}, sh)\}$
return $(Q, \mathcal{F}, \Sigma, Q_f, \Delta)$
```

if it is present at level o, then the non-conflicting symbols at that level are also present at level $o + 1$, for later TA construction. This is automatically ensured for orders o to $o + size - 2$ in the algorithm, but might require an additional push and insert operation for order $o + size - 1$. In addition, LearnOaOp repeatedly shifts and reassigns orders while reinserting conflicting symbols, resulting in $O(|\mathcal{F}|^2)$ time and $O(|\mathcal{F}|)$ additional space where $|\mathcal{F}|$ refers to the size of $\mathcal{F}$. In the above where (PLUS,3) and (STAR,3) are involved in associativity conflicts, the final set O_p is:

```
{ ((SEMI,2),0), ((STAR,3),0), ((SEMI,2),1), ((PLUS,3),1), ((SEMI,2),2) }
```

3.1.3 A Tree Automaton for Inferred Preferences. Based on the preferences specified by the associativity set O_a, the precedence order set O_p, and the input CFG $\mathcal{G}$, Algorithm 3.2 constructs a TA $\mathcal{A} = (Q, \mathcal{F}, \Sigma, Q_f, \Delta)$ that encodes the parsing preferences over $\mathcal{F}$. The set of its states Q is initially populated with states associated with a range of levels, from 0 to the maximum order m in O_p, representing each level in the hierarchy of orders, so $Q \triangleq \{e_0, \ldots, e_m\}$. These *ordered* states are used to generate transitions with precedence relations among symbols as per O_p, with $Q_f \triangleq \{e_0\}$.

Next, we have a *δ-generator* $\delta_{\mathcal{F}}$: $Q \times Q^* \times \mathcal{F} \mapsto \Delta$. The function $\delta_{\mathcal{F}}$ takes the left hand side state, right hand side list of states, and a symbol in $\mathcal{F}$ to produce transition with the symbols in the right positions. In other words, given a transition structure for a symbol in $\mathcal{F}$ obtained from the productions P in G, $\delta_{\mathcal{F}}$ replaces all the nonterminals with the left and right hand states provided, while keeping all the terminal symbols. We write $\overline{e_i}$ to denote an appropriately lengthed list (dictated by the number of nonterminals in the production) filled with the state e_i. For example, given a symbol (IF,4), when we provide $\overline{e_1}$ of length 2 as an input list of states, $\delta_{\mathcal{F}}(e_1, \overline{e_1}, \texttt{(IF,4)})$ returns $\mathtt{e_1} \leftarrow_{\texttt{(IF,4)}}$ IF $\mathtt{e_1}$THEN $\mathtt{e_1}$ which is a result of replacing all the nonterminals from the original production stmt $\rightarrow$IF expr THEN stmt with a state $\mathtt{e_1}$ while keeping all the terminals IF and THEN intact.[3] This procedure of retrieving a template (predefined transition rule) from a map and replacing

[3]In this case, we replace all the nonterminals—stmt, expr, and stmt—with $\mathtt{e_1}$ as it is listed as the only new state to replace the old states and there is no nonterminal associated with the symbols in $\mathcal{F}_{tr}$. Note that if the input list is of length > 1, we

certain (nonterminal) placeholders with provided contents (new nonterminals) is comparable to template-based code generation [31].

For each $(s, i) \in O_\text{p}$, if there exists a pair $(s, p) \in O_\text{a}$ (i.e. s is involved in an associativity conflict), then a transition rule $\delta_\mathcal{F}(e_i, [e_i; \dots e_{i+1}; \dots e_i], s)$ is added to Δ, where there the list of right hand side states include n leading e_i's then e_{i+1} followed by m trailing e_i's. This corresponds to e_{i+1} appearing at a postition p in the right hand side of the production with n leading nonterminals. This ensures that child symbol does *not* appear at the p-th position. For example, `((PLUS,3), 1)` in O_a along with `((PLUS,3), 2)` in O_p adds a rule $\delta_\mathcal{F}(e_2, [e_2; e_3], s) \triangleq \mathtt{e_2} \leftarrow_{(\texttt{PLUS, 3})} \mathtt{e_2\ PLUS\ e_3}$ to Δ.

When a symbol s (at order i) is not contained in O_a then we simply add $\delta_\mathcal{F}(e_i, \overline{e_i}, s)$. For example, `((IF,4), 1)` in O_p results in $\delta_\mathcal{F}(e_1, \overline{e_1}, \texttt{(IF,4)}) \triangleq \mathtt{e_1} \leftarrow_{(\texttt{IF, 4})} \mathtt{IF\ e_1 THEN\ e_1}$ being added to Δ.

Next, for each symbol s' in $\mathcal{F}_\text{tr}$ and its associated state e', we add $\delta_\mathcal{F}(e', [], s')$ to Δ and e' to Q, thus allowing a transition such as $\mathtt{ident} \leftarrow_{(\texttt{IDENT, 1})} \mathtt{IDENT}$ to be included in Δ. Following the addition of rules for the trivial symbols, we connect the states of different levels, by generating $(\epsilon, 1)$-transitions *consecutively*, from e_0 to e_1, e_1 to e_2, and so on until e_{m-1} to e_m.

To handle cycles in the order of symbols correctly, we also need to add transitions that allow for these cycles in the learned automata. Consider symbols s_l and s_h where s_h is at a higher order in O_bp than s_l, with productions $e_l \rightarrow_{s_l} \beta_l$ and $e_h \rightarrow_{s_h} \beta_h$ in $\mathcal{G}$ such that $e_l \in \beta_h$—i.e., s_l can appear deeper in a parse tree than s_h. Intuitively, such circularity in symbol order needs to be reintroduced in the automaton. The function *HighToLow*(G, O_p) identifies such pairs and returns $(s_h, o_h), (s_l, o_l)$ where o_h is the highest order of s_h and o_l is the lowest order of s_l in O_p. For each such pair, we add the transition $\delta_\mathcal{F}(e_{o_h}, \overline{e_{o_l}}, s_h)$ connecting them.

Lastly, the transition rules in Δ along with Q, $\mathcal{F}$, and Q_f result in a TA, as shown in Fig. 7. Given that $|\mathcal{F}|$ is the total number of symbols, GenTA has time and space complexity of $O(|\mathcal{F}|)$. Details of the complexity analysis can be found in the supplementary material [20].

3.1.4 Soundness of Tree Automata Learning. Algorithm 3.2 is sound in the following sense: a tree automaton it constructs faithfully encodes associativity and precedence order, and does not lose unrelated information from the input grammar $\mathcal{G}$.

Given trees T^+ selected by the user, and trees T^- not selected by the user, we define the tree languages: $\mathcal{L}^+ \triangleq \bigcup_{t \in T^+} \mathit{ParseTrees}(t)$ and $\mathcal{L}^- \triangleq \bigcup_{t' \in T^-} P^-(t')$. L^+ is the set of all parse trees described by the selected tree examples, and L^- is the set of trees that are excluded by Greta, which strictly enforces symbol hierarchy as described in Sec. 3.1. These sets can have an overlap, as illustrated in Fig. 10. We want our learned TA to reject all trees described by L^-, which will also exclude this intersected region of L^+. With these definitions, we state our main soundness result.

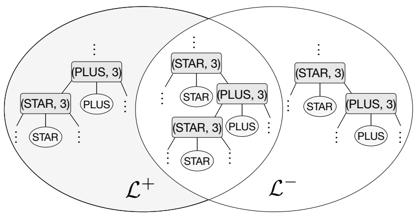


Fig. 10. Trees in $\mathcal{L}^+$ and $\mathcal{L}^-$.

Theorem 3.1 (Soundness of GenTA). *Let $\mathcal{A}_r$ be the finite TA returned by Algorithm 3.2 and $\mathcal{A}_g$ be the TA derived from CFG $\mathcal{G}$, T^- be the negative tree examples. Further, $\mathcal{L}_r = L(\mathcal{A}_r)$, $\mathcal{L}_g = L(\mathcal{A}_g)$, and $\mathcal{L}^- = \bigcup_{t \in T^-} P^-(t)$. Then, $\mathcal{L}_r \supseteq \mathcal{L}_g \setminus \mathcal{L}^-$, and $\mathcal{L}_r \cap \mathcal{L}^- = \emptyset$.*

Proof. Provided in the supplementary material [20]. □

replace each old state (excluding the states associated with trivial symbols) with the new states in the input list starting from the head element consecutively. For example, given structure of `(PLUS, 3)`-transition $\mathtt{expr} \leftarrow_{(\texttt{PLUS, 3})} \mathtt{expr\ PLUS\ expr}$, applying the δ-generator $\delta_\mathcal{F}(e_2, [e_2; e_3], \texttt{(PLUS, 3)})$ produces $\mathtt{e_2} \leftarrow_{(\texttt{PLUS, 3})} \mathtt{e_2\ PLUS\ e_3}$.

```
ss:
| s ss
|  /* empty */
s:
| dcl SEMI
| id EQ x SEMI
| IF LPAREN x RPAREN s
| IF LPAREN x RPAREN s ELSE s
| RETURN x SEMI
| WHILE LPAREN x RPAREN s
| LBRACE ss RBRACE
e:
| e PLUS e
| e DASH e
| e STAR e
| id
| int
| LPAREN e RPAREN

id: IDENT
int: INT
dcl: TINT id EQ e
```

(a) Original grammar

```
ss:
| s1 ss
| /* empty */
s1:
| IF LPAREN e1 RPAREN s1
| s2
s2:
| dcl SEMI
| id EQ e1 SEMI
| WHILE LPAREN e1 RPAREN s2
| RETURN e1 SEMI
| LBRACE ss RBRACE
| IF LPAREN e1 RPAREN s2 ELSE s2
e1:
| e1 PLUS e2 | e2
e2:
| e2 DASH e3 | e3
e3:
| e3 STAR e4 | exp4
exp4:
| id
| int
| LPAREN e1 RPAREN
id: IDENT
int: INT
dcl: TINT id EQ e1
```

(b) Repair via Greta

```
(ss,e1):
| (s,e1) (ss,e1)
(s,e1):
| (if,if) (lparen,lparen) (e,e1)
  (rparen,rparen) (s,e1)
(e2,e2):
| (int,e3)
| (id,e3)
(ss,e1):
| (ϵ,ϵ)
| (ϵ,e2)
(e,e1):
| (e2,ϵ)
| (e2,e2)
| (e2,e4)
| (e2,e3)
(e,e2):
| (e,e2) (dash,plus) (e,e2)
| (e,e2) (dash,dash) (e,e2)
| (e,return) (dash,e2) (e,semi)
| (e,lparen) (dash,e1) (e,rparen)
| (e,e2) (dash,dash) (e,e2)
| (e,lbrace) (dash,e1) (e,rbrace)
(s,e2):
| (return,e2) (exp,plus) (semi,e2)
| (return,e2) (exp,dash) (semi,e2)
| (return,return) (e,e2) (semi,semi)
| (return,lparen) (e,e1) (semi,rparen)
:
```

(c) Repair via classical TA intersection

Fig. 11. Disambiguated grammar returned by Greta vs. simple cross products.

The first statement asserts that $\mathcal{A}_r$ does not lose any unnecessary parse trees in $\mathcal{L}_g$ that are not excluded by the negative examples in T^- (which includes the parse trees selected by the user). The second statement asserts that $\mathcal{A}_r$ does not accept any tree that belongs to $\mathcal{L}^-$, that have been discarded by the user.

We conclude by noting that the CFG translated from the resulted automaton is *not* readily usable as a repair solution of the original grammar. That is because the TA is generated to be permissive enough to include all the original grammar's parse trees that do not have to do with conflicts, but this makes the TA accept even the trees that should not be allowed. For example, a tree for `IF 1 THEN 2 PLUS 3` is accepted by $\mathcal{A}_r$, whereas it is not accepted by $\mathcal{A}_g$. This leads us to discuss our next contribution, an algorithm for intersection of $\mathcal{A}_r$ and $\mathcal{A}_g$.

3.2 Intersecting Tree Automata for Context-Free Grammars

Our intersection algorithm ultimately computes the component-wise cross-products of sets of states, accepting states, and transition rules [10]. In the standard definition of finite tree automata (FTA) intersection, this cross-product construction is applied uniformly to all states and transitions, combining transitions solely based on matching arity of ranked symbols and enumerating all resulting state tuples, regardless of whether they can contribute to accepting run. Algorithm 3.3 departs from the textbook construction in two key aspects that are essential in our grammar-based setting: (1) it gives special treatment to transitions involving terminals (*i.e.*, producing the product of transitions not only based on matching arity, but also the precise sequence of terminals and nonterminals on the right-hand side), and (2) it incorporates reachability-based optimisations directly into the construction. That is because (1) the standard definition does not account for transitions involving terminals, producing a more complex grammar due to states mapped from all the existing terminals. This helps us to produce a more idiomatic grammar, as shown by the

ALGORITHM 3.3: IntersectTA

```
Input: $\mathcal{A}_g = (Q_g, \mathcal{F}, \Sigma, Q_{gf}, \Delta_g)$ and $\mathcal{A}_r = (Q_r, \mathcal{F}, \Sigma, Q_{rf}, \Delta_r)$ tree automata
Output: $\mathcal{A} = (Q, \mathcal{F}, \Sigma, Q_f, \Delta)$ a tree automaton
$Q_f \leftarrow Q_{gf} \times Q_{rf}$
/* Learn $\Delta$ wrt. reachable states */
$Q, Q_{tmp} \leftarrow Q_f$
while $Q_{tmp} \neq \emptyset$ do
    for $(e_g, e_r) \in Q_{tmp}$ do
        $\mathcal{F}_g \leftarrow$ symbols of $e_g$ in $\Delta_g$;  $\mathcal{F}_r \leftarrow$ symbols of $e_r$ in $\Delta_r$ /* Includes symbols reachable by
            $(\epsilon, 1)$-transitions */
        $\mathcal{F}' \leftarrow \mathcal{F}_g \cap \mathcal{F}_r$
        for $s \in \mathcal{F}'$ do
            $\delta_s^g \leftarrow \delta_s$ producing $e_g$ in $\Delta_g$;  $\delta_s^r \leftarrow \delta_s$ producing $e_r$ in $\Delta_r$
            $\Delta \leftarrow \Delta \cup \{\delta_s^g \times \delta_s^r\}$
            $Q' \leftarrow$ reachable states of $(e_g, e_r)$ in $\delta_s^g \times \delta_s^r$
            $Q_{tmp} \leftarrow Q_{tmp} \cup \{q \mid q \in Q'$ and $q \notin Q\} \setminus \{(e_g, e_r)\}$;  $Q \leftarrow Q \cup Q'$
/* Remove duplicate states */
$Q_{dup} \triangleq$ FindDupStates$(Q, \Delta)$
for $((x_g, x_r), (y_g, y_r)) \in Q_{dup}$ do
    $Q \leftarrow Q \setminus \{(y_g, y_r)\}$
    $\Delta \leftarrow \Delta \setminus \{\delta \mid \delta$ producing $(y_g, y_r)\}$
    Replace $(y_g, y_r)$ with $(x_g, x_r)$ in $\Delta$
/* Introduce $\epsilon$-transitions to simplify $\Delta$ */
$L_q \leftarrow$ Ordered list of $Q$ whose $|\delta|$ is smallest to largest
for $i \in [0 .. |L_q|)$ do
    $\Delta_i \leftarrow \{\delta \mid \delta \in \Delta$ producing $i^{th}$ $Q$ in $L_q\}$
    for $j \in [i+1 .. |L_q|)$ do
        $\Delta_j \leftarrow \{\delta \mid \delta \in \Delta$ producing $j^{th}$ $Q$ in $L_q\}$
        if RHS of $\Delta_i \subset$ RHS of $\Delta_j$ then
            $\Delta'_j \leftarrow (\Delta_j \setminus \Delta_i) \cup \{e_j \leftarrow_{(\epsilon,1)} e_i\}$
            $\Delta \leftarrow (\Delta \setminus \Delta_j) \cup \Delta'_j$
return $(Q, \mathcal{F}, \Sigma, Q_f, \Delta)$
```

grammar in Fig. 11b as compared to the results taken from the textbook definition of cross-products in Fig. 11c. We introduced (2) because simply taking cross-products of each component yields a number of states and transitions that cannot reach the final states, which we can avoid computing entirely, by leveraging reachability analysis.

Algorithm 3.3 summarises the algorithm optimised for reachability as well as transitions involving terminals, to efficiently compute the intersection of tree automata. Below, we elaborate on some of its components. First, the algorithm computes a set of accepting states Q_f by taking the cross-product of the sets of accepting states of the input TAs: Q_{gf} and Q_{rf}. Since we are intersecting two TAs, Q_f consists of only one state (*i.e.*, a pair consisting of states respectively from Q_{gf} and Q_{rf}). Next, the algorithm updates a set of states Q and a temporary list of states Q_{tmp} with Q_f. Notice that if we simply take a cross-product of Q_g and Q_r to populate Q, the resulting set Q might contain states that are not reachable, and, thus, don't have to be included in Q. Therefore, we add to Q only those states that can reach the final state in Q_f to result in a TA. We maintain Q_{tmp} as a worklist by adding any states we encounter for the first time and consuming the states whenever we generate transitions producing them. Specifically, for all (e_g, e_r) in Q_{tmp}, we obtain all the transitions corresponding to (e_g, e_r) with labels that are intersection of the ranked symbols which e_g and e_r can respectively take in Δ_g and Δ_r, possibly with ϵ-transitions. Then, we update Q and Q_{tmp} with states that have not been collected by Q and are *reachable* from—*i.e.*, appearing on

ALGORITHM 3.4: FindDupStates

```
Input: $Q$ a set of states, $\Delta$ a set of transitions
Output: $Q_{\mathrm{dup}}$ a set of state pairs
for $e_i \in Q$ do
    $\Delta_i \leftarrow$ Transitions to $e_i$ in $\Delta$
    $\Delta_i \leftarrow$ Replace $e_i$ with $e_{tmp}$ in $\Delta_i$
    for $e_j \in Q \backslash \{e_i\}$ do
        $\Delta_j \leftarrow$ Transitions to $e_j$ in $\Delta$
        $\Delta_j \leftarrow$ Replace $e_j$ with $e_{tmp}$ in $\Delta_j$
        if $\Delta_i = \Delta_j$ then
            $Q_{\mathrm{dup}} \leftarrow Q_{\mathrm{dup}} \cup \{(e_i, e_j)\}$
return $Q_{\mathrm{dup}}$
```

the right-hand sides of the transitions from—$(e_{\mathrm{g}}, e_{\mathrm{r}})$, while we remove $(e_{\mathrm{g}}, e_{\mathrm{r}})$ from Q_{tmp}. These steps are repeated to add unseen states to Q and new set of transitions to Δ until Q_{tmp} is empty.

Upon collecting all the *raw* cross-products of the transitions, we remove any duplicate states, identified via RemoveDupStates in Algorithm 3.4, and their corresponding transitions, to reduce the TA following the idea of TA minimisation [10]. Lastly, we examine Δ again to determine which (sub)set of transitions are repeated for states and introduce `(ϵ, 1)`-transitions to simplify the TA further. Moreover, let $|Q_{\mathrm{g}}|$, $|Q_{\mathrm{r}}|$ and $|\Delta_{\mathrm{g}}|$, $|\Delta_{\mathrm{r}}|$ be the numbers of states and transitions of the two input automata. Then, IntersectTA costs $O((|Q_{\mathrm{g}}| \cdot |Q_{\mathrm{r}}|)^2 \cdot |\Delta_g| \cdot |\Delta_r|)$ time and $O((|Q_{\mathrm{g}}| \cdot |Q_{\mathrm{r}}|)^2 + |\Delta_g| \cdot |\Delta_r|)$ space in the worst case. We include details of complexity analysis in the supplementary material [20].

Theorem 3.2 (Correctness of Greta). *The intersection of automaton $\mathcal{A}_r$ (GenTA's result) with $\mathcal{A}_g$ (the automaton derived from CFG $\mathcal{G}$) produces a tree automaton recognizing the language $\mathcal{L}_g \setminus \mathcal{L}^-$.*

Proof. Follows from Theorem 3.1 and set intersection. □

4 Implementation and Evaluation

We implemented the proposed methodology in a tool called Greta. Greta is written in OCaml and depends on OCaml's standard LR(1) parser generator Menhir [30] for identifying ambiguities in the given grammar. We use Menhir, as it can handle more complex grammars than `ocamlyacc`, an LALR(1) parser generator, and produces more detailed and comprehensible error messages for debugging faulty grammars, making it the parser generator of choice in OCaml [25]. The initial CFG is therefore expressed in Menhir's input format. Then, tree examples are created based on the set of productions that lead to each conflict identified by Menhir.

4.1 Interaction Design

We designed the user interface of Greta in a way that reduces the number of choices the user has to make, so Greta would resolve N ambiguities with *at most* N interactions with the user. We do so by presenting one set of tree alternatives out of the group of involved tokens (*i.e.*, symbols) per state, where a conflict happens, according to Menhir. For example, if a grammar has two precedence order conflicts, one between `(IF,4)` and `(PLUS,3)` and another between `(IF,4)` and `(STAR,3)`, Menhir reports a conflict at a state reached after processing the `(IF,4)`-production where tokens involved are `PLUS` and `STAR`. In this case, Greta presents only one set of trees, each of which alternate the depths of symbols, *e.g.*, `(IF,4)` and `(PLUS,3)`, respectively.

We aim to reduce the interaction burden on the user by *not* requiring them to choose a tree example for every single ambiguity in the grammar. Consider the following scenario where there are three ambiguities: one between `(IF,6)` and `(STAR,3)`, another between `(IF,4)` and `(STAR,3)`, and third one between `(IF,4)` and `(IF,6)`. Based on our interaction design, we present the two sets of trees to the user: one showing trees with alternating depths of symbols `(IF,6)` and `(STAR,3)` and another one with symbols `(IF,4)` and `(STAR,3)`. Suppose the user has selected precedence relations `(IF,6)` > `(STAR,3)` and `(IF,4)` < `(STAR,3)`. Based on this, we can infer `(IF,4)` < `(IF,6)`. Thus, Greta ends up resolving three ambiguities with only two tree selections. Note that if the user has chosen `(IF,6)` > `(STAR,3)` and `(IF,4)` > `(STAR,3)`, Greta would have required another user interaction. Therefore, the number of prompts to the user depends not just on the conflicts in the input CFG, but also on the set of trees selected by the user in each interaction.

4.2 Experimental Setup

A run of Greta involves a series of user prompts, in which the user specifies their preference between two tree examples. Greta then produces a new grammar by applying the methodology from Sec. 3 and provides it as a new input to Menhir. In this process, the user does not have to do or know that Greta involves any operations on TAs. When the newly produced CFG still contains ambiguities, there are subsequent rounds of running Greta involving user interaction until Greta successfully resolves all the ambiguities *or* left with ambiguities not addressable by Greta. We discuss the non-addressable ambiguities in detail in Sec. 4.4. Hence, the end-to-end workflow of Greta involves a series of prompts. Since each prompt provides two tree examples, the cumulative scenarios become exponential in the number of prompts, which quickly gets intractable for manual testing. Our testing framework therefore automates this interaction with Expect [23], a terminal text interface automation tool. Moreover, since we do not involve real users, we present all ambiguities in each grammar and test for all possible scenarios. All our experiments are conducted on a commodity machine running Ubuntu 22.04, with 16GB RAM and 16 logical CPU cores.

Benchmarks. To evaluate our methodology, we accumulated a variety of grammars whose sizes vary *wrt.* numbers of terminals, nonterminals, and productions, as shown in the first three columns in Tab. 1. $G0$ is identical to the running example in Fig. 1. $G1$ is similar to $G0$ but more interesting as it allows symbols for binary operations `(PLUS,3)` and `(STAR,3)` to conflict with symbols `(IF,4)` and `(IF,6)`, making it possible for the grammar to have more ambiguities. We obtained the next three grammars $G2$ to $G4$ from course assignments of a popular class on compiler design. $G2$ describes a simple boolean language, $G3$ and $G4$ define more complex languages including while loop and various binary operations in addition to boolean expressions. $G5$ and $G6$ are collected from questions posted on StackOverflow. $G5$ is a simple grammar written for logical expressions,[4] whereas $G6$ describes a language for expressing constraints with inequalities and binary operations.[5] $G1$ to $G6$ comes in 2 or 3 variants, GNa to GNb (or GNc), containing different number/type of ambiguities to see how Greta performs in terms of accuracy and speed with an increasing number of ambiguities.

We compiled practical grammars of real-world languages: $G7$, $G8$, and $G9$. $G7$ is the Michelson grammar that is used for specifying smart contracts on the Tezos blockchain, while $G8$ is the grammar of Kaitai, a declarative language for describing binary structures in Tezos.[6] $G7$ grammar was obtained from a publicly available subset of the Michelson grammar,[7] combined with more constructs from the Michelson reference.[8] Lastly, $G9$ is a subset of the SQL language, a standard language used for accessing and manipulating databases.

[4]https://stackoverflow.com/questions/910445/issue-resolving-a-shift-reduce-conflict-in-my-grammar.
[5]https://stackoverflow.com/questions/4588397/fixing-lemon-parsing-confilcts?rq=3.
[6]https://gitlab.com/tezos/tezos/-/tree/master/client-libs/kaitai-ocaml
[7]https://github.com/aigarashi/ocaml_of_michelson
[8]https://tezos.gitlab.io/michelson-reference/

Table 1. Aggregate results from Greta runs. The table presents the number of terminals ($|\Sigma|$), the number of nonterminals ($|V|$), the number of productions ($|P|$), the total number of ambiguities in each grammar reported by Menhir ($A_{p,a}$) where p refers to the number of ambiguities *wrt.* precedence order and a refers to the number of ambiguities *wrt.* associativity, the average number of prompts (Δ), time spent in converting the grammar to TA in ms (Conv), time spent for TA learning in ms (Learn), TA intersection time in ms with percentage of scenarios successfully disambiguated as subscript ($I^{\text{def}}_{\%}$), TA intersection time in ms without reachability-based optimisation ($I^{1}_{\%}$), without duplicate removal optimisation ($I^{2}_{\%}$), without epsilon introduction optimisation ($I^{3}_{\%}$), and without all three optimisations ($I^{123}_{\%}$), each with their respective percentage fixed as subscript, total time for manual disambiguation in s (T_{man}), and number of production edits for manual disambiguation (Δ_{man}).

	$\lvert\Sigma\rvert$	$\lvert V\rvert$	$\lvert P\rvert$	$A_{p,a}$	Δ	Conv	Learn	$I^{\text{def}}_{\%}$	$I^{1}_{\%}$	$I^{2}_{\%}$	$I^{3}_{\%}$	$I^{123}_{\%}$	T_{man}	Δ_{man}
$G0$	13	5	10	$5_{2,2}$	4	0.06	1.68	0.62_{50}	3.06_{50}	0.91_{0}	0.50_{20}	2.21_{0}	265	9.5
$G1$a	11	4	10	$4_{3,1}$	4	0.06	0.75	0.84_{46}	3.53_{46}	1.36_{0}	0.67_{15}	3.02_{0}	163	6
$G1$b	11	4	10	$7_{6,1}$	7	0.06	0.93	0.83_{47}	2.42_{47}	0.97_{0}	0.58_{23}	1.85_{0}	202	7
$G1$c	11	3	9	$9_{6,2}$	8	0.05	1.07	0.80_{44}	1.94_{44}	0.89_{0}	0.62_{6}	1.36_{0}	169	9.5
Average				$7_{5,1}$	6.3	0.06	0.92	0.82_{45}	2.63_{46}	1.08_{0}	0.62_{15}	2.08_{0}	178	7.5
$G2$a	10	3	10	$4_{2,2}$	4	0.07	0.72	0.94_{100}	3.15_{100}	1.76_{100}	0.86_{100}	2.97_{100}	294	12
$G2$b	10	3	10	$6_{3,2}$	5	0.06	0.76	0.68_{100}	1.56_{100}	0.83_{100}	0.53_{100}	1.26_{100}	123	8
$G2$c	10	2	9	$12_{6,3}$	9	0.05	1.20	0.83_{100}	1.52_{100}	0.92_{100}	0.62_{100}	1.05_{100}	178	12
Average				$7_{4,2}$	6.0	0.06	0.89	0.82_{100}	2.08_{100}	1.17_{100}	0.67_{100}	1.76_{100}	198	10.7
$G3$a	17	8	20	$2_{1,1}$	3	0.13	1.01	3.39_{50}	10.71_{50}	4.16_{0}	2.97_{11}	8.75_{0}	244	14
$G3$b	18	9	21	$3_{2,1}$	6	0.10	1.01	1.63_{50}	8.23_{50}	2.54_{0}	1.49_{11}	8.09_{0}	209	12.5
Average				$2_{2,1}$	4.5	0.12	1.01	2.51_{50}	9.47_{50}	3.35_{0}	2.23_{11}	8.42_{0}	226	13.2
$G4$a	25	10	26	$3_{1,2}$	3	0.17	1.10	2.64_{100}	15.20_{100}	3.51_{100}	2.63_{100}	8.23_{100}	123	3
$G4$b	25	8	24	$12_{6,3}$	9	0.12	1.36	1.81_{100}	7.47_{100}	2.52_{100}	1.36_{100}	7.05_{100}	176	10.5
$G4$c	25	8	24	$16_{6,4}$	10	0.11	1.45	1.67_{100}	7.58_{100}	2.35_{100}	1.17_{100}	7.13_{100}	154	10
Average				$10_{4,3}$	7.3	0.13	1.30	2.04_{100}	10.09_{100}	2.80_{100}	1.72_{100}	7.47_{100}	151	7.8
$G5$a	8	4	9	$2_{1,1}$	2	0.06	0.47	0.64_{100}	2.09_{100}	0.90_{100}	0.49_{100}	1.93_{100}	80	4
$G5$b	8	4	9	$2_{1,1}$	3	0.06	0.61	0.73_{100}	3.13_{100}	0.82_{100}	0.58_{100}	2.34_{100}	56	5
$G5$c	8	2	7	$6_{3,2}$	5	0.04	0.71	0.49_{100}	0.84_{100}	0.58_{100}	0.41_{100}	0.71_{100}	86	8
Average				$3_{2,1}$	3.3	0.05	0.60	0.62_{100}	2.02_{100}	0.76_{100}	0.50_{100}	1.66_{100}	74	5.7
$G6$a	21	5	23	$2_{1,1}$	2	0.19	1.03	2.24_{100}	4.80_{100}	2.88_{100}	2.33_{100}	5.02_{100}	84	5
$G6$b	21	5	23	$14_{7,4}$	11	0.16	1.75	1.96_{100}	8.22_{100}	2.72_{100}	1.38_{100}	6.59_{100}	220	14.5
$G6$c	21	5	23	$18_{8,6}$	14	0.13	2.03	2.28_{100}	8.32_{100}	3.21_{100}	1.54_{100}	6.71_{100}	266	17
Average				$11_{5,4}$	9.0	0.16	1.60	2.16_{100}	7.12_{100}	2.94_{100}	1.75_{100}	6.11_{100}	190	12.2
$G7$	56	12	77	$3_{2,1}$	3	0.51	2.47	4.79_{100}	24.67_{100}	5.60_{100}	4.01_{100}	19.45_{100}	337	8.5
$G8$	37	32	72	$9_{3,3}$	6	116.19	127.02	1.92_{100}	5828.16_{100}	2.89_{100}	1.51_{100}	242.55_{100}	305	6.5
$G9$	29	18	42	$23_{7,9}$	16	0.24	2.81	5.14_{100}	155.44_{100}	6.45_{100}	3.26_{100}	23.77_{100}	549	25

4.3 Experimental Results

Aggregate results from completed runs of Greta are presented in Tab. 1. Grammars are arranged in an increasing order of size per benchmark source, where the size can be approximated by the numbers of productions $|P|$ as well as terminals $|\Sigma|$, and the nonterminals $|V|$. Variations of each grammar, should there be any, are sorted by the number of ambiguities in an increasing order, as shown by the $A_{p,a}$ column where p is the number related to precedence order and a related to associativity. In addition to the default configuration ($I^{\text{def}}_{\%}$), we report results under several ablations of the intersection algorithm, shown in the $I^{1}_{\%}$, $I^{2}_{\%}$, $I^{3}_{\%}$, and $I^{123}_{\%}$ columns in Tab. 1.[9] These ablations selectively disable optimisations introduced in Algorithm 3.3: (i) $I^{1}_{\%}$ disables computing raw cross-products of all transitions without restricting to reachable states via learned Δ; (ii) $I^{2}_{\%}$ disables duplicate-state removal; (iii) $I^{3}_{\%}$ disables ϵ-introduction; and (iv) $I^{123}_{\%}$ disables all three

[9]The subscripted percentage % refers to success rate under each approach.

optimisations. This breakdown allows us to isolate how each optimisation affects both effectiveness and efficiency of Greta. We address the following research questions to evaluate the results:

- RQ1: How *effective* is Greta in eliminating *all* the ambiguities in the grammars, and what factors contribute to its performance?
- RQ2: How does increase in ambiguities or grammar size affect *efficiency* of Greta?
- RQ3: How *scalable* is Greta *wrt.* the input grammar size or number of ambiguities?

4.3.1 RQ1: Effectiveness. Overall, Greta successfully repairs ambiguous grammars with an average of 85% fix rate. Seven out of ten grammars show a perfect 100% fix rate, with the lowest fix rates from $G1$ for which most failures come primarily from the fact that Menhir, an LR(1) parser, can look ahead only *one* token at a time and is unable to distinguish differences in parsing options that would be clear if it could look ahead further. In other words, there are situations when Greta reports remaining ambiguities in the repaired grammar as it relies on Menhir to identify them, even though there are no ambiguities in the underlying CFG. In such cases, attempts by Greta to remove forbidden trees will not change the grammar further.

These cases are reported as failures, contributing to lower fix rates of $G0$, $G1$ and $G3$ in Tab. 1. For example, in the case of $G0$, if a tree specifying `(IF,6) < (IF,4)` is selected, Greta produces a grammar containing the fragment shown in Fig. 12. Given these productions, suppose Menhir tries to parse an expression like `IF TRUE THEN IF TRUE THEN 1 ELSE 2`. It can be done by first applying the rule `x2 →IF cond THEN x2 ELSE x2`, and then at the nonterminal `x2` after `THEN`, applying either (1) the same rule *or* (2) ϵ-transitioning to `x3` and then applying `x3 →IF cond THEN x3`, even though (1) would produce different number of `ELSE`s in the program. On the other hand, if `(IF,4) < (IF,6)` is selected, generating transitions in Fig. 13, Menhir no longer reports ambiguities as parsing can be done only in one way: first, by applying `x2 →IF cond x2`, then at the nonterminal `x2` after `THEN`, ϵ-transitioning to `x3`, and applying `x3 →IF cond THEN x3 ELSE x3`.

```
x2 → IF cond THEN x2 ELSE x2
x2 → x3
x3 → IF cond THEN x3
  ⋮
```

Fig. 12. Productions *wrt.* `(IF,6) < (IF,4)`

Across all benchmarks, the success rate (%) of $I^1_{\%}$ matches that of the default configuration ($I^{\mathrm{def}}_{\%}$). Although $I^1_{\%}$ performs intersection using unrestricted cross-products, the subsequent duplicate-state elimination and ϵ-introduction steps ensure that the resulting automata and grammars are identical to those produced under all optimisations. This indicates that optimisations 1 primarily improves efficiency rather than correctness. In contrast, disabling duplicate-state removal ($I^2_{\%}$) or ϵ-introduction ($I^3_{\%}$) significantly degrades effectiveness for several grammars, notably $G0$, $G1$s, and $G3$s. In these cases, the resulting tree automata and CFGs remain semantically correct, but contain redundant or overlapping states and nonterminals that render the grammar unsuitable as input to a deterministic LR parser generator. This highlights an important distinction: while tree automata and CFGs tolerate redundancy, deterministic parsing does not. Consequently, optimisations such as duplicate elimination and ϵ-introduction are not merely performance improvements, but are necessary to ensure compatibility with LR parsing, which Greta relies on via Menhir. In addition, when all optimisations are disabled ($I^{123}_{\%}$), these issues compound, leading to consistently low success rates for the same grammar.

```
x2 → IF cond THEN x2
x2 → x3
x3 → IF cond THEN x3 ELSE x3
  ⋮
```

Fig. 13. Productions *wrt.* `(IF,4) < (IF,6)`

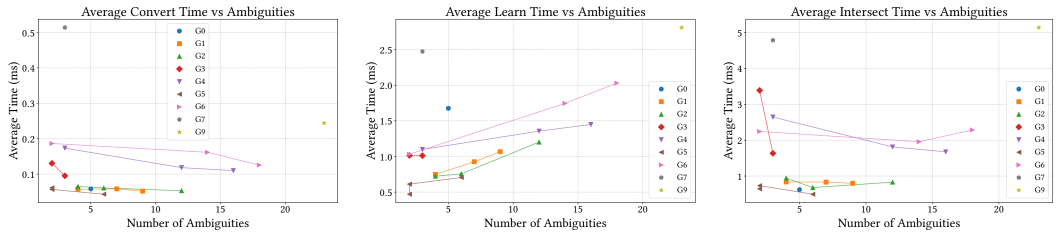
For almost all benchmark data, there is at least one scenario, *i.e.*, a combination of user selections, which leads to elimination of all the ambiguities. This means, Greta can be used to offer parsing options that eventually make the grammar conflict-free, which is potentially useful for the users who are looking for a possible way to disambiguate the grammar. It takes about 6.3 prompts to fully fix a grammar with an average of 8 ambiguities. This shows the benefit of our interaction design, as it indeed requires less than N prompts to address N ambiguities on average.

4.3.2 RQ2: Efficiency. Greta is fast in performing disambiguation across different grammars, with an average of less than 20ms running time (excluding the user interaction time), which is barely noticeable to a user. The time measured refers to an average amount of time it took for successful cases, *i.e.*, when the scenarios being tested result in resolution of all the ambiguities. As shown in Tab. 1, the total runtime is split into three different components, showing the time spent for conversion (Conv), TA learning (Learn), and TA intersection with all optimisations enabled ($I_{\%}^{\text{def}}$). Across most grammars, conversion contributes the least to the overall runtime, followed by learning, with intersection typically dominating. The optimisations made in Algorithm 3.3, bring an algorithm that is usually strictly quadratic in the number of productions, to one that is more efficient. Intersection is slowest for $G7$ which has the most productions, and for which there are many reachable states to compute. Time spent for conversion and intersection appears largely unaffected by the number of ambiguities, as shown in in Fig. 14a and Fig. 14c. Fig. 14b shows more prominent slowdown for learning *wrt.* number of ambiguities, with a clearer upward trend.

$G8$, one of the larger grammars, has a significantly slower learning and conversion time compared to the other grammars. We believe that this is likely due to its larger number of states in combination with a large number of productions, for which book-keeping procedures in conversion and learning in our implementation, scale quadratically. In contrast, the default intersection time for $G8$ remains relatively modest. This is because intersection is computed only over reachable state pairs, which dramatically reduces the effective search space. The impact of this optimisation is confirmed by the ablation results: when reachability-based pruning is disabled ($I_{\%}^{1}$ and $I_{\%}^{123}$), intersection time for $G8$ increases by several orders of magnitude. This shows that the reachability optimisation is important for containing the cost of intersection in grammars with large numbers of states and productions. That is, while large grammars can result in expensive conversion and learning, the intersection optimisations are effective at preventing state explosion from dominating total runtime.

4.3.3 RQ3: Scalability. We examine performance trends as the input grammar size (*i.e.*, the numbers of productions $|P|$, of terminals $|\Sigma|$, and of nonterminals $|V|$) and ambiguity count increases to assess scalability. Across grammar families $G1$-$G6$, increasing the number of ambiguities has only a moderate impact on conversion or intersection time, with a slower-than-linear upward trend in learning time, as can be shown in Fig. 14. This indicates that increasing the number of ambiguities has only a moderate impact on learning time, indicating that Greta scales well with respect to ambiguity count alone. On the other hand, grammars with large numbers of productions and symbols or complex nonterminal structure (*e.g.*, deeply nested or mutually recursive nonterminals) incur higher costs, reflecting the increased size of the induced automata. For instance, even for a grammar with more than 20 ambiguities (*e.g.*, $G9$), Greta successfully eliminates all supported ambiguities within a few milliseconds. Moreover, Greta takes the longest on average to repair $G8$, despite having fewer ambiguities than grammars like $G2$c, $G4$b-c, $G6$b-c and $G9$, due to its large production set combined with its productions containing a large number of mutually recursive nonterminals. These results suggest that scalability of Greta is governed less by the sheer number of ambiguities than by the size and the structural complexity of the grammar.

Although these trends may suggest that Greta scales robustly in ambiguity count and reasonably in grammar size, there is a caveat: the times measured were obtained from successful runs, whereas a bigger and more complex grammar is likely to contain edge scenarios which are outside the scope of Greta or of Menhir that Greta depends on. This implies that scalability of Greta is subject to its scope and limitations which we discuss further in Sec. 4.4.



(a) Average CFG to TA convert time (ms) versus number of ambiguities for different grammar types.

(b) Average TA learning time (ms) versus number of ambiguities for different grammar types.

(c) Average intersection time (ms) versus number of ambiguities for different grammar types.

Fig. 14. Performance metrics for various grammar types across different numbers of ambiguities. Each point represents a grammar variant, with markers indicating the grammar family.

4.3.4 Manual Disambiguation. To assess the effort required to repair grammars *without* Greta, two of the authors independently performed manual disambiguation for each grammar using Menhir alone. Manual disambiguation consisted of repeatedly inspecting Menhir's error messages and conflict reports, editing grammar productions to address reported ambiguities, and re-running Menhir until no further conflicts were reported. Both authors were equally familiar with Menhir's input format and the interpretation of its conflict diagnostics.

For each grammar, we recorded the total time spent on manual repair and the number of production edits made; the values in columns T_{man} and Δ_{man} of Tab. 1 are the averages of the time and of the edits made. In practice, manual repair takes on the order of minutes rather than (milli)seconds and requires dozens of production edits, even for grammars with a modest number of ambiguities. Moreover, manual fixes were occasionally incorrect or overly restrictive, unintentionally eliminating valid parses. Such cases required undoing or revising earlier changes, further increasing the overall effort. A key contributor to this cost is the form in which ambiguities are presented to the user: Menhir reports conflicts at the level of LR automaton states and actions, requiring users to reconstruct the competing parse structures and infer the intended disambiguation before modifying the grammar. An example illustrating the user interfaces of Menhir and Greta is provided in the supplementary [20], highlighting the contrast in how ambiguities are presented to users.

4.4 Scope and Limitations

4.4.1 Menhir vs. Greta: Scope of Ambiguities Addressed. While Greta relies on Menhir's ambiguity detection mechanisms, Menhir reports an over-approximation of ambiguities that arise during LR(1) parser construction, in the form of shift/reduce and reduce/reduce conflicts. These conflicts may be caused by various sources such as operator precedence and associativity, dangling-else ambiguities, grammar underspecification, or LR(1)-specific limitations unrelated to genuine syntactic ambiguity. Out of these conflicts detected by Menhir as LR(1) conflicts, Greta operates on a subset of them: *i.e.*, ambiguities for which parsing preferences can be captured by selecting one of the two alternative tree examples, as described in Sec. 3.1. In other words, not all conflicts reported by Menhir are addressable by Greta, and we discuss these non-addressable conflicts in details in Sec. 4.4.3. Accordingly, Menhir acts as a the conflict detection backend, and Greta focuses on repairing those conflicts that can be resolved via example-driven grammar restrictions.

4.4.2 Termination Behaviour. Greta operates by iteratively repairing the input grammar and re-invoking Menhir on the resulting grammar. The procedure terminates in one of three cases: (i) Menhir reports no remaining conflicts; (ii) the remaining conflicts do not fall within Greta's example-based ambiguity class (Sec. 3.1); or (iii) Menhir reports conflicts that are false positives and cannot be eliminated by further restriction of the grammar. In cases (ii) and (iii), Greta detects that no further progress can be made, halts without attempting further iterations, and reports to the user that the remaining conflicts are not addressable.

The user cannot become stuck in an infinite loop. Whenever Greta applies a repair step, the resulting grammar admits a strict subset of the parse trees admitted by the previous grammar. Since the original grammar yields a finite set of parse trees and conflicts, this monotonic reduction of admissible parse trees guarantees termination. We now discuss the kinds of conflicts that belong to cases (ii) and (iii), and therefore constitute inherent limitations of Greta.

4.4.3 Limitations and Non-addressable Conflicts. Greta is designed to resolve ambiguities that can be expressed as *example-based binary choices* (Sec. 3.1, Sec. 4.4.1), allowing it to offer a lightweight and intuitive interaction model. However, this design choice also imposes inherent limitations on the classes of ambiguities that can be addressed. As described in Sec. 4.4.2, there are two scenarios where Greta may terminate without resolving all reported conflicts: (1) false positives reported due to limitations of Menhir, (2) ambiguities for which tree examples cannot be constructed because the underlying parsing preferences cannot be represented as two alternatives. In both cases, the remaining conflicts fall outside the scope of Greta and are therefore reported as non-addressable.

First, the scenario (1) happens due to Greta's dependency on Menhir, as mentioned in Sec. 4.3.1. A natural way we can address this issue is by employing a parser that is not limited by the number of lookahead tokens. For example, dypgen [26] is a GLR parser generator not limited to a fixed number of lookahead tokens. Alternatively, Earley [28] allows a construction of a parser with an unbounded number of lookahead tokens based on an algorithm that creates a chart of possible parses via dynamic programming. While these are viable options, a potential issue would be their inefficiency (both have cubic time complexity compared to linear time of Menhir), and, as for Earley, the input CFG needs to be formatted with a combination of functions, requiring more work to define a parser. Moreover, scenario (2) can happen when examples cannot be constructed to form trees as per Sec. 3.1. Informally, this can be understood as 3 or more different productions contributing to an ambiguity, making it difficult to show parsing preferences as two different alternatives.

5 Related Work

Grammar repair. Our work was inspired by the approach of Adams and Might [2], where tree regular expressions capturing undesired sets of parse trees are taken as input and are intersected with a TA corresponding to the ambiguous CFG. This results in a procedure whose time complexity grows exponentially in the number of negative examples *and* grammar size (due to negation). In our approach, in addition to providing a simpler interface for the user than having to write formal tree regular expressions, the learning algorithm from Sec. 3.1 constructs a *single* tree automaton encoding all user preferences, and performs a single intersection step which is quadratic in the size of the automata regardless of the number of examples. Additionally, our TA learning step is linear in the number of examples, resulting in a more efficient and scalable approach. Other existing works present ways to resolve grammar ambiguities [11, 17, 33] or repair grammars [29]. Grammars can be repaired with declarative disambiguation rules called *filters* [11, 17, 33] that remove undesired parse trees either as a post-parse step, or embedded in the generated parser. Unlike our approach, filters do not provide a way to update a grammar incorporating the disambiguation rules. This limits the interpretability of the interactions between the parsing restrictions and the original CFG, since one needs to regularly reference disambiguating rules to understand what parse trees are really accepted by a CFG. Another work [29] addresses the problem of repairing the grammar that fails tests from a provided test suite, assuming that a correct grammar exists. It fixes the erroneous grammar by iteratively applying a set of bespoke patches. Greta does not rely on a fixed set of rewrite rules, and instead utilises a general mechanism of TA derived from user-chosen examples.

Tree automata learning via Angluin's algorithm. A well-known approach to learn deterministic finite automata (DFAs) is Angluin's L* algorithm [4, 5], which relies on an oracle, and maintains an observation table through oracle queries, which is later used to construct the DFA. While there is existing work that applies the idea of the L* algorithm for learning a TA [7], we found it difficult to adopt this approach in our setting. This is because the L* algorithm assumes the existence of an oracle that can answer membership queries (*i.e.*, if a tree belongs to a TA), and providing such an oracle would be challenging in our programming by example setting.

We initially considered treating interactions with the user as an oracle, and involve user prompts in the learning loop. This approach, however, turned out to be neither desirable nor necessary because the L* algorithm for tree automata learning requires a representative set of example trees, so that every TA transition exercises at least one example. This would require (a) constructing examples involving symbols that are not involved in any conflicts and (b) numerous interactions with the user in order to correctly simulate an oracle, defeating the purpose of our tool, especially when the size of the given grammar is large. Our learning algorithm, on the other hand, does not need to involve the user other than when they are asked to specify their preferred tree examples *wrt.* ambiguities. Alternatively, we could consider the input CFG—or, the TA translated from the CFG, to be precise—as an oracle. Such an oracle, however, could not correctly answer membership queries about the additional user-provided restrictions.

Programming by example. Programming by example (PBE) entails the synthesis of programs from a set of input-output examples. PBE has been adopted in synthesising grammars in Leung *et al.*'s work [22], which constructs parsers from user-provided parse tree examples. This work encodes parsing restrictions through specification of associativity and precedence order rules, similar to the construction of grammar restriction rules in Greta. These rules are then used to remove unwanted parse trees. This approach requires the user to provide examples that fully characterise the intended language, since the methodology does not permit providing an existing grammar as context. For large languages, or for expanding existing grammars, this can get quickly impractical. Our technique uses an existing grammar as context, with user preferences as additional constraints to produce an updated grammar. This simplifies the process of writing a grammar from scratch, but also allows for incremental updates and maintenance of existing grammars.

Wang *et al.* [35] implement PBE for learning TAs for data completion tasks in tabular data. Similar to other PBE approaches, this requires user-provided examples, constrained by formulae in a domain-specific language (DSL) for reasoning about tabular data. While in Wang *et al.*'s work tree automata are employed for compact representations of the user examples, the problem tackled in that work is fundamentally different from ours in that the approach is specialised for tabular data completion tasks, and relies heavily on the user providing correct and sufficient examples.

6 Conclusion

In this work, we presented a novel take on the problem almost as old as the area of programming languages itself—specifying syntax of a programming language in a way that is deterministic and is free from parsing ambiguities [3]. Specifically, we have cast the problem of disambiguating a context-free grammar as an instance of *programming by example*, structuring the process of grammar repair (*i.e.*, removing ambiguities) as a series of lightweight interactions with the grammar designer, who guides the repair by choosing their preferred parse trees.

We believe that the key idea of our approach—compiling pairs of complementary positive/negative tree examples into tree automata used to refine an initially provided "base" grammar—has applications beyond just resolving ambiguities in context-free grammars. In particular, tree automata-based representation of examples can be used as a tool to guide the design of formal grammars, benefitting both programming language designers and automated tools for grammar learning.

Acknowledgments

We thank Hila Peleg and George Pîrlea for their feedback on drafts of this paper. We also thank the anonymous OOPSLA'26 reviewers. This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair" MOE-MOET32021-0001.

Data Availability

The software artefact accompanying this paper is available online [21]. It contains the OCaml implementation of Greta, including GenTA (Sec. 3.1) and IntersectTA (Sec. 3.2), as well as benchmark data and build scripts for reproducing the evaluation results reported in Sec. 4.

References

[1] Paul W. Abrahams. 1966. A final solution to the Dangling else of ALGOL 60 and related languages. *Commun. ACM* 9, 9 (1966), 679–682. doi:10.1145/365813.365821

[2] Michael D. Adams and Matthew Might. 2017. Restricting grammars with tree automata. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 82:1–82:25. doi:10.1145/3133906

[3] Alfred V. Aho, Stephen C. Johnson, and Jeffrey D. Ullman. 1973. Deterministic Parsing of Ambiguous Grammars. In *POPL*. ACM Press, 1–21. doi:10.1145/512927.512928

[4] Dana Angluin. 1981. A Note on the Number of Queries Needed to Identify Regular Languages. *Inf. Control.* 51, 1 (1981), 76–87. doi:10.1016/S0019-9958(81)90090-5

[5] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Inf. Comput.* 75, 2 (1987), 87–106. doi:10.1016/0890-5401(87)90052-6

[6] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *PLDI*. ACM, 95–110. doi:10.1145/3062341.3062349

[7] Jérôme Besombes and Jean-Yves Marion. 2007. Learning tree languages from positive examples and membership queries. *Theor. Comput. Sci.* 382, 3 (2007), 183–197. doi:10.1016/J.TCS.2007.03.038

[8] Leon Bettscheider and Andreas Zeller. 2024. Look Ma, No Input Samples! Mining Input Grammars from Code with Symbolic Parsing. In *FSE*. ACM, 522–526. doi:10.1145/3663529.3663790

[9] John Boyland and Daniel Spiewak. 2009. TOOL PAPER: ScalaBison Recursive Ascent-Descent Parser Generator. 253, 7 (2009), 65–74. doi:10.1016/J.ENTCS.2010.08.032

[10] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sophie Tison, and Marc Tommasi. 2008. Tree Automata Techniques and Applications. https://inria.hal.science/hal-03367725v1/file/tata.pdf.

[11] Luís Eduardo de Souza Amorim, Michael J. Steindorfer, and Eelco Visser. 2018. Towards Zero-overhead disambiguation of Deep Priority conflicts. *The Art, Science, and Engineering of Programming* 2, 3 (Mar 2018). doi:10.22152/programming-journal.org/2018/2/13

[12] Alexander Demenchuk. 2008. Beaver - a LALR Parser Generator. https://beaver.sourceforge.net/.

[13] Joost Engelfriet. 2015. Tree Automata and Tree Grammars. *CoRR* abs/1510.02036 (2015). arXiv:1510.02036 http://arxiv.org/abs/1510.02036

[14] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining input grammars from dynamic control flow. In *ESEC/FSE*. ACM, 172–183. doi:10.1145/3368089.3409679

[15] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *POPL*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 317–330. doi:10.1145/1926385.1926423

[16] Stephen C Johnson et al. 1975. *Yacc: Yet another compiler-compiler*. Vol. 32. Bell Laboratories Murray Hill, NJ.

[17] Paul Klint and Eelco Visser. 1994. Using filters for the disambiguation of context-free grammars. In *Proc. ASMICS Workshop on Parsing Theory*. Milan, Italy, 1–20.

[18] Vu Le and Sumit Gulwani. 2014. FlashExtract: a framework for data extraction by examples. In *PLDI*. ACM, 542–553. doi:10.1145/2594291.2594333

[19] Xuan-Bach Dinh Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *ESEC/FSE*. ACM, 593–604. doi:10.1145/3106237.3106309

[20] Yunjeong Lee, Gokul Rajiv, and Ilya Sergey. 2026. Grammar Repair using Examples and Tree Automata: Extended Version. *CoRR* abs/2602.18166 (2026). http://arxiv.org/abs/2602.18166

[21] Yunjeong Lee, Gokul Rajiv, and Ilya Sergey. 2026. *Greta: Grammar Repair using Examples and Tree Automata (Artefact)*. doi:10.5281/zenodo.18654374 Working repository available at https://github.com/verse-lab/greta.

[22] Alan Leung, John Sarracino, and Sorin Lerner. 2015. Interactive parser synthesis by example. In *PLDI*. ACM, 565–574. doi:10.1145/2737924.2738002

[23] Don Libes. 1995. *Exploring expect - a Tcl-based toolkit for automating interactive programs (2. ed.)*. O'Reilly.

[24] José Nuno Macedo and João Saraiva. 2020. Expressing disambiguation filters as combinators. In *SAC*. ACM, 1348–1351. doi:10.1145/3341105.3374123

[25] Anil Madhavapeddy and Yaron Minsky. 2022. *Parsing with OCamllex and Menhir* (2 ed.). Cambridge University Press, Cambridge, UK, Chapter 19, 361–373.

[26] Emmanuel Onzon. 2012. dypgen: User's Manual. http://dypgen.free.fr/.

[27] Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming not only by example. In *ICSE*. ACM, 1114–1124. doi:10.1145/3180155.3180189

[28] Christophe Raffalli and Rodolphe Lepigre. 2020. earley: Parsing library based on Earley Algorithm. https://opam.ocaml.org/packages/earley/.

[29] Moeketsi Raselimo and Bernd Fischer. 2021. Automatic grammar repair. In *SLE*. ACM, 126–142. doi:10.1145/3486608.3486910

[30] Yann Régis-Gianas and François Pottier. 2016. Menhir Reference Manual. https://gallium.inria.fr/~fpottier/menhir/.

[31] Eugene Syriani, Lechanceux Luhunu, and Houari A. Sahraoui. 2018. Systematic mapping study of template-based code generation. *Comput. Lang. Syst. Struct.* 52 (2018), 43–62. doi:10.1016/J.CL.2017.11.003

[32] Mark van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. 2002. Disambiguation Filters for Scannerless Generalized LR Parsers. In *CC (LNCS, Vol. 2304)*. Springer, 143–158. doi:10.1007/3-540-45937-5_12

[33] Eelco Visser. 1997. A Case Study in Optimizing Parsing Schemata by Disambiguation Filters. In *International Workshop on Parsing Technologies, IWPT*, Sandiway Fong (Ed.). 210–224. https://aclanthology.org/1997.iwpt-1.24/

[34] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *PLDI*. ACM, 452–466. doi:10.1145/3062341.3062365

[35] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of data completion scripts using finite tree automata. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 62:1–62:26. doi:10.1145/3133886

[36] R. Michael Wharton. 1976. Resolution of Ambiguity in Parsing. *Acta Informatica* 6 (1976), 387–395. doi:10.1007/BF00268139

[37] Shanchan Wu, Jerry Liu, and Jian Fan. 2015. Automatic Web Content Extraction by Combination of Learning and Grouping. In *WWW*. ACM, 1264–1274. doi:10.1145/2736277.2741659

Received 2025-10-10; accepted 2026-02-17